

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



# Solr

## 实战

[美] Trey Grainger 著  
Timothy Potter 著

范炜 等译

### Solr in Action

Yonik Seeley  
为本书作序



# Solr 实战

[美] **Trey Grainger**  
**Timothy Potter** 著

范炜 等译

Solr in Action

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

## 内 容 简 介

本书介绍了当下最流行的开源搜索技术解决方案 Solr。在搜索引擎视域下,循序渐进地介绍了 Solr 是什么、Solr 能做什么,以及如何更好地使用 Solr 进行开发。在搜索基础层,本书从 Solr 的快速搭建入手,介绍了 Solr 背后的信息检索基本概念,之后重点讲解了构建一个搜索引擎所需的核心模块:索引构建、文本分析、执行搜索及处理搜索结果。在搜索功能层,详细介绍了 Solr 的四大增强型搜索功能:分面搜索、搜索结果高亮、查询建议、搜索结果分组等。在搜索研究的进阶层,介绍了 SolrCloud、多语种搜索及复杂查询操作等。最后,围绕搜索引擎的本质核心问题“相关度”展开了讨论与展望。

本书适合搜索技术工程师、搜索应用设计者以及对搜索引擎技术感兴趣的读者阅读,也可作为高校计算机专业信息技术方向、信息管理与信息系统专业等的课程参考资料。

Original English Language edition published by Manning Publications, USA. Copyright © 2013 by Manning Publications. Simplified Chinese-language edition copyright © 2017 by Publishing House of Electronics Industry. All rights reserved.

本书简体中文版专有出版权由 Manning Publications 授予电子工业出版社。

未经许可,不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字:01-2014-4906

## 图书在版编目(CIP)数据

Solr 实战/(美)崔·格兰杰(Trey Grainger),(美)提摩斯·波特(Timothy Potter)著;范炜等译.  
—北京:电子工业出版社,2017.5

书名原文:Solr in Action

ISBN 978-7-121-31165-9

I. ①S… II. ①崔… ②提… ③范… III. ①搜索引擎—程序设计 IV. ①TP391.3

中国版本图书馆CIP数据核字(2017)第060465号

策划编辑:符隆美

责任编辑:徐津平

印 刷:三河市良远印务有限公司

装 订:三河市良远印务有限公司

出版发行:电子工业出版社

北京市海淀区万寿路173信箱 邮编:100036

开 本:787×980 1/16 印张:39.25 字数:791千字

版 次:2017年5月第1版

印 次:2017年5月第1次印刷

定 价:129.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888,88258888。

质量投诉请发邮件至 zltz@phei.com.cn,盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819,faq@phei.com.cn。

# 译者序

---

搜索无处不在，搜索已经融入我们工作、生活的方方面面。除每天使用的通用搜索引擎以外，几乎我们使用的所有互联网应用、ERP、企业内联网等都提供（或应该具备）了搜索功能。我们向搜索引擎发出查询请求后，很快就能得到一大堆相关的搜索结果，它是怎么做到的呢？这是搜索引擎的黑箱秘密。开源搜索技术 Solr & Lucene 是一套很好的技术工具，能够帮助我们打开这一黑箱，洞悉搜索引擎背后的技术原理与运行机制，并能够借此“真枪实干”地做搜索。

一种观点认为，搜索引擎是信息检索系统的一种类型，即面向网络环境中的海量异构数据的采集、组织与检索。在学科专业领域中，信息检索的研究一直以来都是象牙塔里的一门高深学问。Solr & Lucene 的出现拉近了信息检索学术界与搜索引擎业界之间的距离，加速了搜索应用的设计与开发，新的模型算法与搜索功能在 Solr 的开发技术架构中得以快速转化与实现。

信息检索是“用户需求（行为）—搜索技术（引擎）—信息资源（集合）”三位一体的复合研究主题。搜索研究的原始起点是用户需求与搜索行为，搜索引擎技术是将用户需求与信息资源进行相关度匹配的重要中介手段。没有搜索技术就无从着手开发，不关注用户需求与搜索行为的话，搜索就无从谈起。那么，如何全面理解搜索并开发搜索应用？为了回答这个问题，《Solr 实战》的内容安排试图在技术、资源与用户三者之间找到一种平衡。本书以用户搜索问题解决为导向，通过各种操作实例，全面系统化地介绍了 Solr 的主要功能与使用方法。由于搜索引擎技术体系庞大，涉及数据采集（网络爬虫）、文本处理、自然语言处理、索引、搜索及其他

许多方面,要把这么多理论内容和实践操作放进一本书里,着实不是一件容易的事情,然而本书作者做到了!本书虽然也算得上是一本大部头,但相较于很多大部头技术书,作者对技术细节与技术复杂度的收放考虑还是比较得当的,同时也兼顾涵盖了分布式 SolrCloud、多语种搜索、个性化推荐等热门主题,具有较好的搜索基础普适性与进阶助推性作用。

如果把搜索引擎比作一台汽车, Lucene 与 Solr 的关系就好比是“发动机”与“造车平台”。Lucene 提供了核心的索引与搜索功能, Solr 向下对 Lucene 进行了底层技术封装,隐藏了大量技术细节,降低了进入搜索应用开发的技术门槛。与此同时, Solr 向上扩展了开放集成的大量高级搜索功能,用来满足各种搜索业务需求。因此, Solr 为我们提供了一个更易于学习、操作与应用的全功能开源搜索平台。面对现实的搜索业务需求,从 Solr 入手可以让你进入搜索技术应用层的快车道,通过对 Solr 的深入了解与搜索业务结合,在解决实际搜索问题的大前提下,继而回归到 Lucene 内核的学习。当遇到诸如索引效率问题、相关度模型与算法问题时,自然而然地就深入到 Lucene 内部了。所以,从搜索应用设计与开发的角度来看,从 Solr 入门是一条更有实践效率和学习成就感的途径。

当前开源搜索技术研究与实践非常活跃,这直接反映在了 Solr 的版本变化上。2010 年 3 月 Solr 开源项目与 Lucene 开源项目合并之后, Solr 成为 Lucene 的一个子项目。从 3.1 版本起, Lucene 与 Solr 的版本保持了一致。版本更新在索引与搜索性能优化、多元化搜索功能引入、交互体验设计、bug 修复等方面不懈地努力着,但 Solr 搜索技术的核心模块并没有发生实质性改变。所以,从搜索技术的学习角度来说,书中使用的 Solr 版本是够用的。Solr 5.0 之后涉及少量基础操作命令的变化,但这不影响书中内容的讲解,我们在书中必要的地方做了注释,方便读者对比。当然,寄希望于读完一本书就能胜任搜索开发恐怕是不现实的。面对网上各种碎片化、不同版本的 Solr 技术资料,对于初涉搜索技术的入门者而言,以这本大部头作为搜索技术学习的小目标,会是一个不错的选择。

本书的初译按章节分工如下:范炜负责第 1、5、6、7、8、16 章,侯任夷负责第 2、3 章,金国栋负责第 4、12 章,邹婧琳负责第 9、10 章,陈皇丹负责第 11、13 章,吴亚平负责第 14、15 章,张功卫、伍志鹏负责附录与其他翻译资料的整理。最终由范炜进行全书的统稿与校对,胡康林参与了校样的审读。

本书的翻译工作历时较长,在此期间王中英与符隆美两位编辑给予了充分信任与理解,并付出了大量辛苦工作,对此表示衷心感谢。鉴于译者专业水平有限,以及专业术语的不一致等问题,译文中难免存在不当之处,恳请读者批评指正。

范炜

2017 年 3 月 4 日于川大江安河畔



# 原书推荐序

---

Solr 拥有悠久而成功的历史，Solr 4 和 SolrCloud 开启了全新的篇章。《Solr 实战》的出版非常应景。书中包含清晰的案例、启发性的图表，涵盖了从核心概念到最新功能的方方面面，有助于你快速掌握 Solr。

2004 年，CNET 科技资讯网（现在的 CBS 互动媒体公司）的搜索引擎服务提供商不再提供服务，CNET 需要一套搜索替代方案，于是有了 Solr。我开始编写 Solr 的时候，虽然不具备专业搜索背景，但做起来很上手。这可能与我的软件设计理念“要快”有关。这一观点有助于 Solr 在传统企业搜索市场以外得以拓展。

截至 2005 年年底，Solr 为 CNET 旗下的许多网站提供了搜索与分面导航功能，而且很快成为了开源软件。2006 年 1 月，Solr 被捐赠给了 Apache 软件基金会，成为 Lucene PMC 的子项目，与 Lucene Java（现在的 Lucene Core）是兄弟项目关系。由于 Solr 使用 Lucene 作为全文搜索核心库，两者的技术开发人员有较大的重合度，2010 年两个项目就合并在一起了。现在 Lucene 和 Solr 仍然可以分别下载，但它们的开发由一个团队统一进行管理。Solr 版本号跳跃式地匹配了 Lucene 版本号，因而两者新版本的发布得以同步。

Solr 4 版本是 Solr 的一个重要里程碑，增加了 SolrCloud。SolrCloud 拥有一组高度可扩展的功能，包括无单点故障的分布式索引构建。NoSQL 方面的功能扩展包括事务日志、更新持久性、乐观并发与原子更新。Trey 和 Timothy 作为 Solr 的资深专家和社区成员，他们撰写的《Solr 实战》一书涉及了 Solr 重要的新功能，并为 Solr 新手提供了很好的起步指引。

现在 Solr 的应用比我预想的要多得多, 涉及图书馆集成系统、电子商务平台、数据分析与商务智能产品、内容管理系统及互联网搜索, 等等。Solr 从早期只有少数采用者逐渐成长为大规模的全球社区, 在用户的帮助和志愿者的积极推动下, 不断向前发展。

《Solr 实战》介绍了 Solr 使用的必备知识和技术, 这些都是自 2004 年以来的开发积累。本书在手, 可以助你掌握 Solr 的开发。你也可以加入到 Solr 的全球社区, 推动 Solr 更好地发展。

Yonik Seeley  
Solr 的创造者

# 前言

---

2008 年，我受命接管凯业必达（CareerBuilder）招聘网的搜索技术团队。最初使用微软的 FAST 搜索平台，但随后我们意识到，搜索对凯业必达招聘网而言非常重要，与其继续依赖搜索服务提供商，不如在团队内培养搜索专家。我随即开始调研同类型的开源搜索软件，发现 Solr 似乎满足我们搜索开发所需的大部分核心功能。2009 年夏，经过充分的搜索技术准备之后，我们决定将已有的搜索系统转换到 Solr。

Solr 的发展很顺利。Solr 构建在开源搜索库 Lucene 之上。2005 年 2 月，Lucene 正式成为 Apache 顶级项目。2006 年 Solr 被捐赠给 Apache 软件基金会，并于 2007 年 1 月成为 Apache 顶级项目。这两项技术的发展都达到了一定规模之后，在 2010 年 3 月合并为一个项目。

2010 年夏，我们的搜索平台完全转换到了 Solr。在转换过程中，我们提升了搜索速度，大幅度减少了搜索架构所需的服务器数量，避免了昂贵的软件许可费用，增强了平台的稳定性，从先前依赖搜索服务提供商的外部主导转向内部的搜索自主创新。

当时我们并没有意识到搜索自主创新所带来的附加价值。我们已经能够开发出一整套全新的搜索应用产品，包含关键词搜索、语义搜索、大数据分析 & 实时推荐引擎。我们将 Solr 作为可扩展的搜索架构，一小时内通过数百台服务器处理数十亿级的文档与数百万级的查询请求。我们进入了灵活扩展的云服务时代，在数据爆炸中寻求数据的社会意义与价值。Solr 让我们有能力解决迎头而来的挑战。

当 Manning 出版社找到我谈《Solr 实战》一书的写作事宜时，我犹豫了，因为我清楚这是一项艰巨的任务。我提出一个要求：需要一位得力的合作者，而



Timothy Potter 正是合适的人选。Tim 拥有多年 Lucene 和 Solr 搜索方案的开发经验。他利用 Solr 和其他大数据前沿技术，在社交数据的文本分析系统构建与实时分析解决方案方面有着丰富经验。多年来，我俩都得到过 Solr 社区的诸多帮助。当得知开源社区确实需要一本案例驱动的 Solr 指南时，Tim 和我很高兴能撰写《Solr 实战》这本书，为下一代搜索工程师提供帮助。这本书是五年前我们开始接触 Solr 时希望看到的，不论你是刚开始学习 Solr，还是需要补充搜索知识，希望这本书能够对你有帮助。

# 致谢

---

与 Solr 一样，这本书的问世离不开广大开源社区成员的支持，在此对他们表示感谢。

- Lucene/Solr 提交者不但能编写令人惊叹的代码，还能提供宝贵的专业知识和建议，同时又能对社区新成员保持足够的耐心。
- 感谢 Lucene/Solr 社区的活跃成员积极贡献代码、更新 Wiki 和其他文档，并回答 Lucene 和 Solr 邮件列表问题。
- 感谢 Solr 的创造者 Yonik Seeley 为本书作推荐序。
- 感谢那些通过 Manning 图书早期访问计划 MEAP 阅读本书草稿，在作者在线论坛上发表评论的读者。
- 感谢在本书写作过程中提供宝贵反馈的评阅人：Alexandre Madurell、Ammar Alrashed、Brandon Harper、Chris Nauroth、Craig Smith、Edward Welker、Gregor Zurowski、John Viviano、Leo Cassarani、Robert Petersen、Scott Anthony、Sopan Shewale 及 Uma Maheshwar Rao Gunuganti。
- 感谢 Ivan Todorović 与 John Guthrie 在本书付印之前的短短时间里对本书进行了详细的技术校对。
- 感谢 Manning 出版社的相关责任编辑们：Elizabeth Lexleigh、Susan Conant、Melinda Rankin、Elizabeth Martin 和 Janet Vail。
- 感谢 Manning 出版社的 Bert Bates 帮助我们提高写作中的讲解说明水平。
- 感谢家人与朋友在我们漫长的研究与写作过程中给予支持。

## Trey Grainger

首先要感谢我了不起的妻子 Lindsay，在这本书漫长的写作过程中离不开她的支持与耐心。如果没有她的理解和帮助，这本书不可能完成，特别是在这期间我们的女儿出生了。

我还要感谢 Paula 和 Steven Woolf，他们花了不少时间照看 Melodie，才让我得以顺利完成书稿。最后，我要感谢凯业必达公司团队，包括公司领导层和我的搜索团队，感谢你们让我有机会与优秀的人一起工作，一起构建一个领先的搜索平台，以此惠及社会。

## Timothy Potter

我要感谢我的母亲 Sharon Russom，她从小就培养我对学习和书籍的热爱。还要感谢我的父亲 David Potter 对我的学业与工作的支持。没有 Lori Joy 的帮助，这本书不可能完成。感谢你对我在深夜和周末加班的理解与支持，并在写作早期宣传此书。

还要感谢我之前效力的 Dachis 集团。如果没有他们对 Solr 提出那些富有见地的问题，并给我机会去使用 Solr 构建一个大规模搜索解决方案的话，我不可能完成这本书。

# 关于本书

---

无论是处理大数据、构建云服务或开发多租户模式的 Web 应用，拥有一套快速且可靠的搜索解决方案至关重要。Apache 的 Solr 是基于 Lucene 的、可扩展的、部署待命的一种开源全文搜索引擎。它提供了开箱即用的多语种关键词搜索、分面搜索、智能匹配、内容聚类和相关度加权等核心搜索功能。

《Solr 实战》是使用 Apache 的 Solr 实现快速且可扩展的搜索的一份权威指南。书中精心准备的案例既有基础的关键词搜索，也有扩展为支持数十亿文档与查询的搜索系统。通过学习本书，你会深入了解如何使用 Solr 的核心功能，例如，搜索结果的分面导航、匹配到的结果片段高亮、字段折叠与搜索结果分组、拼写检查、查询自动补全、函数查询等。你还会了解如何提升 Solr 使用技能，进阶主题包括大规模生产用例、复杂多语种搜索、复杂查询操作及相关度调整的高级策略。

## 本书框架

本书共分三个部分：“初识 Solr”“Solr 的核心功能”与“Solr 进阶”。对于新接触 Solr 与搜索的一般读者，强烈建议你们按顺序阅读第 1 部分（第 1~6 章），这些章节中出现的许多概念是前后衔接、彼此关联的。

第 2 部分（第 7~12 章）的内容是大多数搜索应用最常见的功能。你可以根据当下的需求关注点，跳过第 2 部分中与需求无关的章节，这样做不会影响其他内容的学习。例如，搜索结果分组是许多搜索引擎的常见功能，但是如果你的数据不需要分组，那么你就可以安全地跳过第 11 章。

第 3 部分（第 13~16 章）的内容有一定难度，介绍了一些高级主题，包括多语种搜索、在大规模集群环境中运行 Solr、高级数据操作及相关度调整等。

书中多数章节都有动手操作，帮助你更好地掌握知识点。我们对每个例子的要求是，既易于操作又能覆盖章节主题。许多例子使用的数据来自真实世界的数据集，以便你接触到现实用例的处理。

第1章介绍 Solr 可以处理的数据类型和应用场景。你会了解 Solr 可以解决的问题类型，从大体上了解 Solr 的主要功能。Solr 4 是 Lucene/Solr 项目的一个重要里程碑，所以即使你对 Solr 先前的版本很熟悉，我们还是建议你阅读第1章，了解一下 Solr 4 带来的全新的、激动人心的功能。

第2章介绍如何在本地工作站上安装和运行 Solr。启动 Solr 之后，我们演示了如何使用 Solr 对一组示例文档集进行索引和搜索。另外，我们还简要介绍了 Solr 的 Web 管理控制台。

第3章介绍了搜索的基础理论，以及 Solr 是如何实践这些理论的。本章最吸引人的内容是倒排索引和相关度评分原理。相关度得分决定了在搜索结果顶部呈现哪些最相关的文档。即使你使用过 Solr，我们还是建议你阅读一下这一章，加深对搜索引擎基本操作的理解。

第4章介绍 Solr 配置的基础知识，主要讲解 Solr 的主配置文件：solrconfig.xml。这一章介绍 Solr 最重要的一些配置设定，特别是影响 Solr 如何处理客户端应用请求的那些配置。这一章学到的知识会用在书中其他章节。

第5章介绍 Solr 如何索引文档，从另一个重要的配置文件 schema.xml 谈起。你会学习通过定义字段来表示结构化数据，例如，数字、日期、价格和唯一标识符等。这一章还会介绍如何使用 solrconfig.xml 对更新请求进行处理和配置。

第6章建立在第5章的内容基础之上，介绍如何使用文本分析索引文本字段。Solr 被设计用于对需要全文搜索的文档进行有效地搜索与排序。文本分析是搜索过程的重要组成部分，它消除了索引文本和查询之间的语言差异。

读到此处，你就已经具备了 Solr 的基础知识，下一步是使用 Solr 来满足你自己的搜索需求。随着对搜索与 Solr 的了解不断加深，你的需求会超越基础的关键词搜索，进而去实现一些常见的搜索功能，例如，高级查询解析、搜索结果高亮、拼写检查、自动建议、分面搜索及搜索结果分组等。

第7章介绍如何构造与执行查询。你会了解 Solr 的许多查询解析器，以及如何对搜索结果进行排序、格式响应、返回及调试等。

第8章介绍 Solr 最强大和最受欢迎的功能之一——分面搜索。Solr 的分面搜索提供了搜索条件限定，通过对搜索结果进行归类，帮助用户发现更多信息。

第9章介绍如何在搜索结果中高亮显示查询词项，以改善搜索解决方案的用户体验。

第10章介绍拼写检查与自动建议。当用户输入几个字符之后，Solr 的自动建议

功能就会给出一个与用户的输入相关的查询建议列表。

第 11 章探讨 Solr 的搜索结果分组与字段折叠功能。当索引包含许多类似的文档时，这样做有助于返回最佳的搜索结果组合，例如，一座城市中同一家餐厅的多个分店。

第 12 章帮助你为在生产环境中部署 Solr 做好准备。这一章会帮助你规划硬件与资源需求，以及决定是否需要考虑使用分片与复制来处理大量的文档与查询请求。

第 13 章介绍 Solr 的分布式功能，也就是 SolrCloud。你会学习如何以云模式运行 Solr，通过对搜索应用进行扩容，以应对大规模用户与文档。读完这一章，你会对 Solr 通过多个服务器上的分布式索引以达到可扩展性与容错性有充分的了解。

第 14 章是第 6 章文本分析内容的延伸，教会你如何在搜索引擎中处理多语种文本。如果需要处理索引中的非英语文本或支持多语种，这一章是必读的。

第 15 章探讨了高级查询功能，包括函数查询、地理空间搜索、多层级分面透视。以及跨文档与跨索引的连接。

第 16 章介绍搜索结果相关度改进技术，例如，调整权重、基于函数的评分、其他相似度算法及相关度得分调试等。此外，进一步讨论了使用 Solr 进行个性化搜索与推荐。

本书有三个附录，对前几章的一些子主题进行了更深入的讨论。附录 A 介绍如何使用 Solr 代码库，为满足特定功能需要和修复官方版本的错误，定制自己的 Solr 分发版本。这个附录是第 12 章开头内容的延伸。

附录 B 以表格形式列出了 Solr 开箱即用的多语种配置。这个附录是第 14 章语种配置内容的扩展。

附录 C 更详细地介绍了数据导入处理器（Data Import Handler, DIH），扩展了第 10 章和第 12 章的相关内容，演示了几种大型公开可用数据集的导入步骤。

## 如何使用本书

《Solr 实战》适合所有软件工程师阅读，无须具备搜索引擎从业经验。本书内容由浅入深，兼顾基础知识与高阶主题，即使经验丰富的 Solr 专业人士也能从最后几章中收获新知。虽然整本书（英文原著）页数超过 600 页，但其中包含了有趣的和实践性的真实案例，在理论与实践之间做出了有效平衡。不论对于刚接触 Solr 还是已有多年 Solr 应用经验的读者，这本书都有一定参考价值。

如前所述，第 1 部分是书中其他部分的基石，这部分内容对 Solr 新手来说非常重要。按顺序阅读这些章节，可以快速有效地掌握 Solr 与搜索技术基础。如果你是 Solr 新手，第 2 章会向你展示如何第一次启动和使用 Solr，第 3 章会介绍搜索的核心理论知识，这些知识是书中其他部分的基础。配置 Solr 服务器并设置字段类型，

以便正确地分析文本内容，这些是掌握 Solr 基本原理的必备知识。

如果你的工作不涉及讨论到的某些功能，可以直接跳过第 2 部分的很多章节。特别是，第 9 章、第 10 章和第 11 章都是比较独立的主题，并不影响阅读后面几章。因此，如果短期内不考虑实现搜索结果高亮、查询建议或搜索结果分组 / 字段折叠，那么可以跳过这三章。第 7 章和第 8 章介绍了许多搜索应用具备的一些常见功能，因此建议你浏览一遍。

其余章节介绍了 Solr 的一些高级主题，解决一些颇有难度的挑战，包括服务器集群扩容、多语种搜索、复杂查询操作以及高级的相关度改进技术。第 2 部分和第 3 部分的章节都基于第 1 部分的内容，第 13 章（SolrCloud）基于第 12 章（搭建 Solr 生产环境），第 15 章（复杂查询操作）基于第 7 章（执行查询和处理搜索结果），第 8 章（分面搜索）与第 16 章（精通相关度）又基于第 15 章。为从本书中获益最大化，请不要跳过作为基础的前几章，它们为后面高级主题的理解提供了必要的知识铺垫。

许多章节包含可执行的示例，你可以一边阅读一边操作。这些示例在介绍新主题的同时，也为你提供了动手探索 Solr 功能的机会。一般来说，你需要做的仅仅是，通过浏览器访问正在运行的 Solr 服务器。你不必运行所有示例，在许多情况下可以直接将它们作为参考配置。不过，运行这些示例会让你获得动手实践经验，这可能有助于理解一些有难度的主题。

是否通读这本书来实现从 Solr 新手到 Solr 专家的进阶，这由你自己决定。如果你暂时不打算通读，当你的兴趣和对 Solr 高级功能的需求增长时，可以随时参考这本书。

## 代码约定及下载

书中的 Java 代码、配置片段、可执行的命令、文件内容与服务器请求 / 响应（以下统称为“源代码”）均采用等宽字体，以便与周围的普通文本区分开。在许多代码清单中，源代码的关键点都被加以注释。某些源代码被设为粗体等宽字体来进行强调。通过添加换行符和缩进方式来格式化源代码，使其适合书中可用的页面空间。不过，有时还是会遇到很长的代码行，这时会在折行处添加行连续符 ➞。

在整本书中，你会找到 Solr 自带的和书中示例用到的文件来源。文件名除了在源代码中引用时使用斜体外，在其他时候仍使用等宽字体。

书中有很多源代码示例，其中较长的代码清单有明确的标题，较短的代码清单则直接出现在正文中间。书中所有的源代码示例都可以从出版商网站 [www.manning.com/SolrInAction](http://www.manning.com/SolrInAction) 或 [www.manning.com/grainger](http://www.manning.com/grainger) 下载。

源代码的根文件夹中附带一个 README.txt 文件，它详细介绍了如何编译与运行这些示例。由于 Lucene/Solr 项目采用的是 Java 语言，所以我们选择 Java 作为本

书的开发语言。对读者而言，一致的编程语言更容易被接受。

第2章下载并安装 Solr 之后，书中其他部分会使用 `$SOLR_INSTALL` 来表示 Solr 的安装目录。与之类似，`$SOLR_IN_ACTION` 表示源代码下载并解压的文件夹。当看到这两个变量时，请在你的系统里替代成实际的文件夹名称。

## 作者在线

购买《Solr 实战》可以免费访问 Manning 出版社的内部网络论坛。你可以对本书发表评论、咨询技术问题，并从作者和其他读者那里获得帮助。要访问并订阅该论坛的话，使用浏览器打开 [www.manning.com/SolrinAction](http://www.manning.com/SolrinAction) 或 [www.manning.com/granger](http://www.manning.com/granger)。该页面介绍了如何登录论坛、可用的帮助种类及论坛使用规则。

Manning 出版社承诺为读者提供一个场所，方便读者之间、读者与作者之间开展有意义的对话交流。论坛不对作者的参与程度做任何承诺，作者对论坛的贡献是无偿的自愿行为。我们建议读者向作者提出一些具有挑战性的问题，以保持他们有兴趣继续留在论坛上。

本书在版流通期间，从出版商网站上可以访问作者在线论坛和查看之前的讨论归档。

## 读者服务

轻松注册成为博文视点社区用户 ([www.broadview.com.cn](http://www.broadview.com.cn))，扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在【提交勘误】处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **与我们交流**：在页面下方【读者评论】处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/31165>





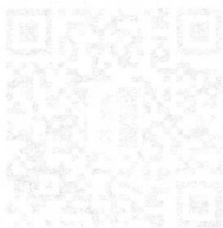
## 关于封面

《Solr 实战》的封面插图为一位哥特妇女，即来自哥特部落的妇女。哥特人是北方人，2000 年前从斯堪的纳维亚半岛来到欧洲，最初定居在波罗的海。他们曾在罗马帝国的衰落期与中世纪欧洲出现时期扮演了重要角色。他们最终分化成两个分支。西哥特人成为罗马人的联邦，然后西迁到法国和西班牙；东哥特人则迁到意大利北部、巴尔干岛及远东黑海。随着时间推移，他们被定居的地区所同化，自己的语言和文化逐渐消失。

这幅图像摘自 Balthasar Hacquet 的《图说西南及东汪达尔人、伊利里亚人和斯拉夫人》(*Images and Descriptions of Southwestern and Eastern Wenda, Illyrians, and Slavs*)，该书由克罗地亚斯普利特民族博物馆于 2008 年重印出版。Hacquet (1739—1815 年) 是奥地利医生与科学家，在奥地利帝国的许多地方，以及威尼托、朱利安阿尔卑斯山和西巴尔干地区长期从事植物学、地质学和民族志学研究，曾与许多不同部落与民族的人一起生活过。Hacquet 发表的许多论文与出版的书籍中附带了手绘插图。

Hacquet 作品中的图画多种多样，生动地反映了 200 年前阿尔卑斯山和巴尔干地区的独特性与个性。在那个时期，两个村庄仅相隔几英里，根据穿着情况就可以分辨出是哪个村庄的人。民族部落、社会阶层或贸易的成员都可以通过他们的穿着进行辨认。穿着样式和地域特色自那时起发生变化。当时的穿着那么多姿多彩，现如今已经消失了。现在很难区分不同大陆的居民，波罗的海、地中海与黑海沿海城镇村庄的居民与欧洲其他地区的居民没什么太大的区别。

Manning 出版的计算机图书封面采用两个世纪前的生活服装插图（如本书封面插图），旨在宣扬创造性、主动性与趣味性。



# 目录

---

## 第 1 部分 初识 Solr..... 1

### 1 Solr 入门..... 2

1.1 为什么需要搜索引擎 .....	3
1.1.1 管理以文本为中心的数据 .....	3
1.1.2 常见的搜索引擎用例 .....	6
1.2 Solr 是什么 .....	8
1.2.1 信息检索引擎 .....	9
1.2.2 灵活的模式管理 .....	11
1.2.3 Java Web 应用 .....	12
1.2.4 一台服务器上的多个索引 .....	13
1.2.5 可扩展性（插件） .....	13
1.2.6 可伸缩性 .....	14
1.2.7 容错性 .....	15
1.3 选择 Solr 的理由 .....	15
1.3.1 面向软件架构师的 Solr .....	15
1.3.2 面向系统管理员的 Solr .....	16
1.3.3 面向 CEO 的 Solr .....	17
1.4 功能概述 .....	17
1.4.1 用户体验功能 .....	17
1.4.2 数据建模功能 .....	19

1.4.3 Solr 4 的新功能 .....	20
1.5 本章小结 .....	22
<b>2 Solr 上手 .....</b>	<b>23</b>
2.1 开始上手 .....	24
2.1.1 Solr 的安装 .....	24
2.1.2 启动 Solr 的示例服务器 .....	25
2.1.3 了解 Solr 主目录 .....	29
2.1.4 对示例文档进行索引 .....	30
2.2 一切都关乎搜索 .....	31
2.2.1 Solr 查询表单详解 .....	31
2.2.2 Solr 的搜索返回机制 .....	34
2.2.3 排名检索 .....	36
2.2.4 分页和排序 .....	36
2.2.5 扩展的搜索功能 .....	38
2.3 Solr 管理控制台一览 .....	40
2.4 根据需求改造搜索示例服务器 .....	42
2.5 本章小结 .....	43
<b>3 Solr 基础理论 .....</b>	<b>45</b>
3.1 搜索、匹配与找寻内容 .....	46
3.1.1 何为文档 .....	46
3.1.2 基本搜索问题 .....	47
3.1.3 倒排索引 .....	50
3.1.4 词项、短语与布尔逻辑 .....	51
3.1.5 找到文档集 .....	53
3.1.6 短语查询与术语位置 .....	55
3.1.7 模糊匹配 .....	57
3.1.8 快速小结 .....	61
3.2 相关度 .....	61
3.2.1 默认相似度 .....	62
3.2.2 词项频次 .....	63
3.2.3 反向文档频次 .....	64
3.2.4 词项权重 .....	65

3.2.5	规范化因子 .....	66
3.3	查准率与查全率 .....	67
3.3.1	查准率 .....	67
3.3.2	查全率 .....	68
3.3.3	达到平衡 .....	69
3.4	搜索的规模化 .....	70
3.4.1	非规范化文档 .....	70
3.4.2	分布式搜索 .....	72
3.4.3	集群 vs. 服务器 .....	73
3.4.4	Solr 的局限 .....	74
3.5	本章小结 .....	75
<b>4</b>	<b>配置 Solr .....</b>	<b>77</b>
4.1	solrconfig.xml 文件概览 .....	80
4.1.1	常见的 XML 数据结构和数据类型元素 .....	82
4.1.2	配置文件更新的应用 .....	82
4.1.3	Solr 的其他配置 .....	83
4.2	查询请求处理 .....	85
4.2.1	请求处理简介 .....	86
4.2.2	搜索处理器 .....	88
4.2.3	Solr 的 browse 请求处理器示例 .....	90
4.2.4	利用搜索组件扩展查询处理 .....	94
4.3	管理搜索器 .....	98
4.3.1	新建搜索器 .....	99
4.3.2	新搜索器预热 .....	100
4.4	缓存管理 .....	103
4.4.1	缓存原理 .....	103
4.4.2	过滤器缓存 .....	105
4.4.3	查询结果缓存 .....	108
4.4.4	文档缓存 .....	110
4.4.5	字段值缓存 .....	110
4.5	其他配置选项 .....	110
4.6	本章小结 .....	111

<b>5 创建索引 .....</b>	<b>112</b>
5.1 微博搜索应用示例 .....	113
5.1.1 面向搜索的内容表示 .....	113
5.1.2 Solr 索引构建概览 .....	115
5.2 设计自己的 schema .....	117
5.2.1 文档粒度 .....	117
5.2.3 索引字段 .....	119
5.2.4 存储字段 .....	119
5.2.5 schema.xml 概览 .....	120
5.3 在 schema.xml 中定义字段 .....	121
5.3.1 必备字段属性 .....	122
5.3.2 多值字段 .....	123
5.3.3 动态字段 .....	124
5.3.4 复制字段 .....	127
5.3.5 唯一键字段 .....	129
5.4 结构化非文本字段类型 .....	129
5.4.1 字符串字段 .....	130
5.4.2 日期字段 .....	131
5.4.3 数值字段 .....	133
5.4.4 高级字段类型属性 .....	134
5.5 发送文档到 Solr 进行索引 .....	137
5.5.1 使用 XML 或 JSON 进行文档索引 .....	137
5.5.2 使用 SolrJ 客户端库添加文档索引 .....	140
5.5.3 向 Solr 导入文档的其他工具 .....	142
5.6 更新处理器 .....	143
5.6.1 将文档提交到索引 .....	145
5.6.2 事务日志 .....	146
5.6.3 原子更新 .....	148
5.7 索引管理 .....	151
5.7.1 索引存储 .....	151
5.7.2 索引片段合并 .....	154
5.8 本章小结 .....	156

<b>6 文本分析</b>	<b>157</b>
6.1 微博文本分析	158
6.2 基础文本分析	161
6.2.1 分析器	163
6.2.2 分词器	163
6.2.3 分词过滤器	164
6.2.4 StandardTokenizer	164
6.2.5 使用 StopFilterFactory 移除停用词	165
6.2.6 使用 LowerCaseFilterFactory 对词项进行小写转换	166
6.2.7 通过 Solr 分析表单进行文本分析测试	167
6.3 为微博文本自定义一个字段类型	169
6.3.1 使用 PatternReplaceCharFilterFactory 折叠重复的字母	172
6.3.2 保留主题标签、提及符号和连字符词项	173
6.3.3 使用 ASCIIFoldingFilterFactory 移除变音符号	177
6.3.4 使用 KStemFilterFactory 提取词干	177
6.3.5 在查询阶段使用 SynonymFilterFactory 加入同义词	178
6.3.6 把过滤器组合在一起	179
6.4 高级文本分析	182
6.4.1 高级字段属性	182
6.4.2 各语种文本分析	183
6.4.3 使用 Solr 插件扩展文本分析	185
6.5 本章小结	188

## 第 2 部分 Solr 的核心功能 191

<b>7 执行查询和处理搜索结果</b>	<b>192</b>
7.1 Solr 请求详解	193
7.1.1 请求处理器	193
7.1.2 搜索组件	197
7.1.3 查询解析器	200
7.2 查询解析器的使用	201
7.2.1 指定查询解析器	201
7.2.2 局部参数	201
7.3 查询和过滤器	204

7.3.1	fq 和 q 参数 .....	204
7.3.2	处理代价过高的过滤器 .....	207
7.4	默认查询分析器 (Lucene 查询解析器) .....	209
7.4.1	Lucene 查询解析器语法 .....	209
7.5	处理用户查询 (eDisMax 查询解析器) .....	215
7.5.1	eDisMax 查询解析器概述 .....	216
7.5.2	eDisMax 查询参数 .....	216
7.5.3	搜索多个字段 .....	216
7.5.4	查询与短语的权重调整 .....	217
7.5.5	字段别名 .....	219
7.5.6	可访问字段 .....	221
7.5.7	最小匹配 .....	221
7.5.8	eDisMax 的优缺点 .....	223
7.6	其他有用的查询解析器 .....	225
7.6.1	字段查询解析器 .....	225
7.6.2	词项查询解析器和原始查询解析器 .....	225
7.6.3	函数查询解析器和函数区间查询解析器 .....	226
7.6.4	嵌套查询和嵌套查询解析器 .....	226
7.6.5	调整权重查询解析器 .....	227
7.6.6	前缀查询解析器 .....	228
7.6.7	空间查询解析器 .....	228
7.6.8	连接查询解析器 .....	228
7.6.9	分支查询解析器 .....	229
7.6.10	外围查询解析器 .....	229
7.6.11	最大得分查询解析器 .....	230
7.6.12	折叠查询解析器 .....	230
7.7	返回搜索结果 .....	231
7.7.1	选择响应格式 .....	231
7.7.2	选择返回字段 .....	233
7.7.3	搜索结果分页 .....	235
7.8	搜索结果排序 .....	237
7.8.1	按字段排序 .....	238
7.8.2	按函数排序 .....	239
7.8.3	模糊排序 .....	239

7.9	调试查询结果 .....	240
7.9.1	返回调试信息 .....	240
7.10	本章小结 .....	241
<b>8</b>	<b>分面搜索 .....</b>	<b>242</b>
8.1	搜索结果概览 .....	243
8.2	建立测试数据 .....	246
8.3	字段分面 .....	250
8.4	查询分面 .....	255
8.5	区间分面 .....	257
8.6	基于分面值的过滤 .....	260
8.6.1	在分面上使用过滤器 .....	260
8.6.2	基于分面值的安全过滤方法 .....	264
8.7	多选分面、键与标记 .....	266
8.7.1	键 .....	266
8.7.2	标记、排除和多选分面 .....	268
8.8	超越分面基础 .....	271
8.9	本章小结 .....	271
<b>9</b>	<b>搜索结果高亮 .....</b>	<b>272</b>
9.1	高亮简介 .....	273
9.2	高亮工作原理 .....	274
9.2.1	为 UFO 目击数据创建新的 Solr 内核 .....	275
9.2.2	索引构建前预处理 UFO 目击数据集 .....	275
9.2.3	探索 UFO 目击数据集 .....	278
9.2.4	开箱即用的高亮 .....	278
9.2.5	高亮具体细节 .....	281
9.2.6	改善高亮显示结果 .....	287
9.3	使用 FastVectorHighlighter 组件提升性能 .....	292
9.4	PostingsHighlighter 组件 .....	293
9.5	本章小结 .....	296
<b>10</b>	<b>查询建议 .....</b>	<b>297</b>
10.1	拼写检查 .....	298



10.1.1	索引维基百科的文章 .....	298
10.1.2	拼写检查举例 .....	300
10.1.3	拼写检查搜索组件 .....	303
10.2	自动建议查询词 .....	309
10.2.1	自动建议请求处理器 .....	309
10.2.2	自动建议搜索组件 .....	311
10.3	文档字段值建议 .....	312
10.3.1	使用 n-grams 生成建议 .....	312
10.3.2	n-gram-driven 请求处理器 .....	314
10.4	基于用户活动提供查询建议 .....	315
10.5	本章小结 .....	320
<b>11</b>	<b>结果分组 / 字段折叠 .....</b>	<b>321</b>
11.1	结果分组 vs. 字段折叠 .....	322
11.2	忽略重复文档 .....	322
11.3	搜索结果中每组返回多个文档 .....	330
11.4	按照函数和查询对结果分组 .....	334
11.4.1	按照函数进行分组 .....	334
11.4.2	按照查询进行分组 .....	336
11.5	对分组结果进行分页和排序 .....	337
11.6	分组陷阱 .....	340
11.6.1	根据结果分组进行分面操作 .....	340
11.6.2	分布式结果分组 .....	342
11.6.3	返回扁平化列表 .....	343
11.6.4	按多值和分词字段进行分组 .....	343
11.6.5	分组性能 .....	344
11.7	使用折叠查询解析器进行高效的字段折叠 .....	344
11.8	本章小结 .....	346
<b>12</b>	<b>搭建 Solr 生产环境 .....</b>	<b>347</b>
12.1	编写一份 Solr 的分发版 .....	347
12.2	部署 Solr .....	348
12.2.1	编译自定义的 Solr 分发版 .....	348
12.2.2	在应用程序中内嵌 Solr .....	349

12.3	硬件和服务配置	350
12.3.1	内存和固态硬盘	350
12.3.2	JVM 设置	351
12.3.3	索引切换	352
12.3.4	实用 Solr 系统配置技巧	355
12.4	数据获取策略	357
12.5	分片和复制	361
12.5.1	分片策略	361
12.5.2	复制策略	364
12.6	Solr 内核管理	368
12.7	管理服务器集群	374
12.7.1	负载均衡器和 Solr 健康检查	374
12.7.2	通用配置 vs. 自定义配置	375
12.8	Solr 的查询与交互	378
12.8.1	REST API	378
12.8.2	可用的 Solr 客户端库	378
12.8.3	使用 SolrJ	379
12.9	监控 Solr 的性能	383
12.9.1	Solr 的插件 / 统计页	383
12.9.2	Solr 缓存性能	387
12.9.3	从请求处理器和 MBeans 获取统计信息	388
12.9.4	外部监控选项	389
12.9.5	Solr 日志	390
12.9.6	加载测试	390
12.10	不同 Solr 版本之间的升级	391
12.11	本章小结	392

## 第 3 部分 Solr 进阶.....393

## 13 SolrCloud.....394

13.1	SolrCloud 上手	395
13.1.1	在云模式下启动 Solr	395
13.1.2	SolrCloud 架构的驱动因素	400
13.2	核心概念	405

13.2.1	集合 vs. 内核.....	405
13.2.2	ZooKeeper .....	406
13.2.3	确定分片和副本的数量 .....	410
13.2.4	集群状态管理 .....	411
13.2.5	确定分片代表 .....	412
13.2.6	SolrCloud 的重要配置.....	413
13.3	分布式索引 .....	416
13.3.1	将文档分配给分片 .....	417
13.3.2	添加文档 .....	418
13.3.3	近实时搜索 .....	421
13.3.4	节点恢复过程 .....	422
13.4	分布式搜索 .....	423
13.4.1	多阶段查询流程 .....	423
13.4.2	分布式搜索的局限性 .....	425
13.5	集合 API.....	425
13.5.1	创建集合 .....	426
13.5.2	集合别名 .....	429
13.6	基本系统管理任务 .....	431
13.6.1	配置更新 .....	432
13.6.2	滚动重启 .....	432
13.6.3	重启故障节点 .....	433
13.6.4	节点 X 处于活跃状态吗 .....	433
13.6.5	新增副本 .....	434
13.6.6	异地备份 .....	434
13.7	高级主题 .....	435
13.7.1	自定义散列 .....	435
13.7.2	分片分割 .....	436
13.8	本章小结 .....	438
<b>14</b>	<b>多语种搜索 .....</b>	<b>439</b>
14.1	为什么语种分析很重要 .....	440
14.2	词干提取 vs. 词形还原 .....	441
14.3	词干提取实战 .....	442
14.4	处理边界情况 .....	447

14.4.1	KeywordMarkerFilterFactory.....	448
14.4.2	StemmerOverrideFilterFactory.....	448
14.5	Solr 支持的语种库.....	449
14.5.1	特定语种的分析器.....	449
14.5.2	基于词典的词干提取 (Hunspell).....	452
14.6	在多语种中搜索内容.....	453
14.6.1	每种语言一个独立字段.....	453
14.6.2	每个语种构建单独的索引.....	459
14.6.3	支持多语种的单个字段.....	462
14.6.4	创建一个字段类型来处理支持多语种的单个字段.....	463
14.7	语种识别.....	475
14.7.1	语种识别更新处理器.....	475
14.7.2	在一个字段中动态分配语种检测分析器.....	482
14.8	本章小结.....	488
<b>15</b>	<b>复杂查询操作.....</b>	<b>489</b>
15.1	函数查询.....	490
15.1.1	函数语法.....	490
15.1.2	函数的搜索.....	492
15.1.3	以字段形式返回函数.....	494
15.1.4	函数排序.....	495
15.1.5	Solr 的可用函数集.....	496
15.1.6	自定义函数.....	502
15.2	地理空间搜索.....	507
15.2.1	搜索附近的一个点.....	507
15.2.2	高级地理空间搜索.....	513
15.3	分面透视.....	523
15.4	引用外部数据.....	526
15.5	跨文档和跨索引的连接.....	528
15.6	使用 Solr 做大数据分析.....	531
15.7	本章小结.....	532
<b>16</b>	<b>精通相关度.....</b>	<b>533</b>
16.1	相关度调整的影响.....	534

16.2	相关度计算的调试 .....	535
16.3	提升相关度 .....	541
16.3.1	字段提升 .....	541
16.3.2	词项提升 .....	543
16.3.3	负载提升 .....	544
16.3.4	函数提升 .....	545
16.3.5	词项邻近度提升 .....	547
16.3.6	提升重要文档的相关度 .....	549
16.4	可插拔的相似度的类实现 .....	552
16.5	个性化搜索与推荐 .....	553
16.5.1	搜索 vs. 推荐 .....	554
16.5.2	基于属性的匹配 .....	554
16.5.3	分层匹配 .....	556
16.5.4	更多类似结果 .....	558
16.5.5	基于概念的匹配 .....	563
16.5.6	地理位置的匹配 .....	568
16.5.7	协同过滤 .....	569
16.5.8	混合方式 .....	573
16.6	塑造个性化搜索体验 .....	574
16.7	开展相关度实验 .....	574
16.8	本章小结 .....	577
附录 A	与 Solr 代码库打交道 .....	578
附录 B	语种字段类型配置 .....	587
附录 C	有用的数据导入配置 .....	593

# 第1部分

## 初识Solr

前6章的重点是探索 Solr 的两个最重要的功能：索引数据和执行查询。阅读完第1部分之后，你应该对 Solr 的查询与索引功能有充分的了解，包括如何对文本和其他类型的数据进行分析，以及在这些数据上进行搜索。

面对每个新主题，我们首先都要从基础开始——学习如何安装 Solr，并在本地运行它。

如果你是初次接触全文搜索，可能会对一些术语感到陌生，那么把第3章当成词典吧。搜索引擎与数据库之间的主要区别是什么？什么是倒排索引？什么是相关度排序以及 Solr 是如何实现的？

掌握了基础知识之后，在第4章中我们将揭开 Solr 的引擎罩，了解如何执行查询请求，熟悉那些对请求进行控制处理的配置设定。Solr 的主要配置文件是 `solrconfig.xml`，它包含许多设置，其中有一些（如缓存管理设置）在刚上手时非常有用，还有一些则适合高级用户。

只有真正地索引了一些文档，才会发现搜索引擎的有趣之处。第5章和第6章介绍如何索引文档，涵盖文档模式设计、字段类型与文本分析。理解索引的核心方面有助于学习本书中的其他内容。

# Solr 入门

## 本章要点

- 搜索引擎处理的数据所具备的特征
- 常见搜索引擎用例
- Solr 核心组件
- 选择 Solr 的理由
- 功能概述

快速发展的技术，如社交媒体、云计算、移动应用与大数据，令人兴奋且充满挑战，一切皆计算的时代到来了。软件架构所面临的一个主要挑战是处理由广泛的全球用户群体消费和产生的大规模数据。此外，用户希望在线应用始终是可用且可响应的。为解决现代 Web 应用的可扩展性和可用性需求，统称为 NoSQL（Not only SQL，不仅仅是 SQL）的非关系型数据存储与处理技术在业内受到了越来越多的关注。这些系统拥有共通的设计模式，是面向特定数据类型的存储与处理引擎，而非强制所有数据遵循曾经的标准化关系模型。换句话说，优化了的 NoSQL 技术用以解决特定数据类型的特定问题。规模需求导致了由各种 NoSQL 和关系型数据库构成的复合架构，一套通吃的数据处理解决方案已成历史。

本书介绍 Apache 的 Solr，它是一种具体的 NoSQL 技术。与其非关系型弟兄一样，Solr 被用来解决特定问题。具体而言，Solr 是可扩展的、开箱即用的企业级搜索引擎，用来搜索大规模文本数据并根据相关度排序结果。这听起来有点绕口，让我们分解开来看：

- 可扩展——Solr 通过集群中多台服务器的分布式运行实现扩展。
- 开箱即用——Solr 是开源的，易于安装和配置，并提供预先配置好的示例服务器，方便上手。
- 为搜索优化——Solr 速度很快，以亚秒级速度执行复杂查询，往往只需花费几十毫秒。
- 大规模文档——Solr 可以用于处理包含百万级文档的索引。
- 以文本为中心——Solr 针对自然语言文本搜索进行优化，如电子邮件、网页、简历、PDF 文档、社交消息（推文和博客）。
- 根据相关度对结果排序——Solr 根据文档与用户查询的相关程度对文档进行排序，并以此顺序返回结果文档。

在本书中，你将学到如何使用 Solr 设计与实现可扩展的搜索解决方案。首先你将了解到 Solr 支持的数据类型和用例，这有助于你掌握 Solr 在现代应用架构蓝图中所处于的位置及其所能解决的问题。

## 1.1 为什么需要搜索引擎

作为本书的读者，相信你对于为什么需要搜索引擎已经有自己的想法了。与其猜测你为什么考虑 Solr，不如来解决你的数据和用例是否需要搜索引擎这个困难的选择问题。这个问题最后可以归结为理解你的数据和用户，并为其选择一种可用的技术。让我们先来看看搜索引擎擅长处理的数据属性。

### 1.1.1 管理以文本为中心的数据

现代应用架构的一个突出特点是与数据搭配相应的存储与处理引擎。如果你是程序员，你应该知道根据数据处理的算法来选择最佳的数据结构，举例来说，当快速随机查找时不使用链表结构。此原则也适用于搜索引擎。类似 Solr 这样的搜索引擎擅长处理的数据表现为 4 个主要特征：

1. 以文本为中心
2. 读主导
3. 面向文档
4. 灵活的模式

第 5 个可能的特征是大量数据需要处理，也就是“大数据”。我们现在讨论的焦点是搜索引擎与其他 NoSQL 技术的不同之处，所以这里没有强调 Solr 可以处理大量数据。

具备上述 4 个主要特征的数据是 Solr 这样的搜索引擎能够有效处理的，但也只



能把它们当作粗略的指南，而非严格的规则。接下来让我们进一步了解上述每一个特征对搜索的重要性。这里，我们将重点放在抽象概念上，具体怎样做会在后续章节深入讲解。

### 以文本为中心

在描述由搜索引擎处理的数据类型时，你必然会遇到非结构化这一术语。我们认为非结构化有些模糊，因为基于人类语言的文本文档都有隐含的结构。你可以将非结构化视为计算机视角的产物，从计算机眼中看到的文本是字符流。字符流必须使用特定语言规则来抽取结构，并使其可搜索。这正是搜索引擎要做的。

我们认为以文本为中心更适合描述 Solr 处理的数据类型，因为搜索引擎是专门用于将文本的隐含结构抽取到索引中，从而改善搜索的。以文本为中心的数据意味着，文档中的文本包含用户在找寻时感兴趣的信息。当然，搜索引擎也支持非文本数据，如日期和数字，但它的主要优势还是处理基于自然语言的文本数据。

如果用户对文本中的信息不感兴趣，搜索引擎可能就不是解决问题的最佳方案了，所以以文本为中心的这个中心很重要。我们来看一个由雇员创建的差旅费用报告的应用。每个报告包含一些结构化数据字段，如日期、费用类型、币种及金额。此外，每项花销可能包含一个备注字段，可以简要描述花销事宜。这是一个包含文本但不是以文本为中心的数据示例，它与会计部门每月生成费用报表时需要搜索备注字段的情况不同。数据包含文本字段并不意味着就与搜索引擎自然适配。

考虑一下你的数据是否以文本为中心吧，主要考虑数据中的文本字段是否包含用户想要查询的信息。如果是，搜索引擎可能是个不错的选择。第 5 章和第 6 章将会使用 Solr 的文本分析能力解锁文本中的结构。

### 读主导

搜索引擎擅长处理的数据的另一个关键特征是读主导，即能实现有效读取，且无须经常更新的数据。对了，你还需要知道，Solr 是对索引中的已有文档进行更新的。“读主导”可以理解为，文档被读取的次数远多于被创建和更新的次数。然而，这也不代表不能写入大量数据，或限制写入新数据的频次。事实上，Solr 4 的一个关键特性是近实时（Near Real-Time, NRT）搜索，允许每秒索引数千文档，并且几乎立时就能搜索到它们。

读主导数据背后的关键点是，当向 Solr 写入数据时，要做好读取和重复读无数次的准备。搜索引擎擅长执行查询数据（读操作），而非存储数据（写操作）。另外，如果必须对搜索引擎里的已有数据经常更新，那么搜索引擎可能不是最佳解决方案。如果需要对已有数据进行快速随机写操作的情况，那么其他的 NoSQL 技术（如 Cassandra）会是一个不错的选择。

## 面向文档

截至目前，我们一直在讨论数据，但现实中搜索引擎处理的是文档。在搜索引擎中，文档是字段的自包含集合，每个字段仅包含数据而不包含嵌套字段。换句话说，Solr 这样的搜索引擎中的文档是平面结构，而且不依赖于其他文档。在 Solr 中，“平面”的概念稍有放宽，一个字段可以有多个取值，但字段不包含子字段。在单一字段中可以存储多个值，但不能在其他字段里嵌套字段。

Solr 面向文档的平面结构方法能很好地处理各种文档格式的数据，如网页、博客、PDF 文件，那么如何对关系型数据库中的规范化数据进行建模呢？在这种情况下，需要对分布在多个表中的数据去规范化，将其组织成一个平面、自包含文档结构。我们将在第 3 章学习转换处理方法。

还要考虑到，文档的哪些字段必须存储在 Solr 中，哪些应该存储在其他系统（如数据库）中。除非数据对搜索或显示结果有用，否则不适合存储在搜索引擎里。例如，对在线视频创建索引时，二进制视频文件不应在 Solr 中存储，大型二进制字段应该存储在其他系统，如内容分发网络（Content-Distribution Network, CDN）。通常，为了满足搜索需求，应该存储的是索引的每个文档信息的最小集合。视频例子清楚地说明了不要将 Solr 作为通用数据存储技术，Solr 的任务是找到兴趣相关的视频，而不是管理大型二进制文件。

## 灵活的模式

Solr 这样的搜索引擎擅长处理的数据的最后一个主要特征是具有灵活的模式。也就是说，索引的文档不必拥有统一的结构。在关系数据库中，表中的每一行都具有相同的结构。在 Solr 中，文档可以有不同的字段。当然，同一索引的文档字段之间可能存在一些重叠，但它们不必是相同的。

假设有一个房屋出租或销售的搜索应用，搜索列表有一些共用的字段，如位置、卧室个数和浴室个数，除此之外，根据房屋类型还可以拥有各自不同的字段。例如，销售的房屋拥有价格和年财产税等字段，出租的房屋拥有月租和宠物管理政策等字段。

总之，通用搜索引擎和 Solr 擅长处理具备这 4 种特征的数据：以文本为中心、读主导、面向文档和灵活的模式。这也说明 Solr 不是一种通用的数据存储和处理技术。

对复杂数据存储与处理的正确认识是无须寻找全能技术。搜索引擎擅长特定数据处理，但不擅长其他技术。这就意味着，在多数情况下，Solr 更多地是对关系型数据库与 NoSQL 数据库进行补充，而非取代它们。

以上介绍了 Solr 擅长处理的数据类型，接下来介绍以 Solr 为代表的搜索引擎适合处理的主要用例。这些用例有助于理解搜索引擎与其他数据处理技术的差异。

### 1.1.2 常见的搜索引擎用例

在本节中，我们来看看用以 Solr 为代表的搜索引擎能做些什么。上一小节讨论了搜索引擎擅长处理的数据类型，这只是指南，而不是严格的规则。在深入细节之前应该清楚一点，要把搜索做到极致是很困难的。现在的用户习惯于 Google 和 Bing 这样的搜索，它们能够快速且有效地满足当前网络信息获取的需求。此外，知名的网站都拥有强大的搜索解决方案，以帮助访客更快地找到信息。对类似 Solr 的搜索引擎进行评估并设计自己的搜索解决方案时，请优先考虑用户体验。

#### 基础关键词搜索

显而易见，搜索引擎支持关键词搜索，这是它的主要功能。其中的原因值得一提：关键词搜索是用户使用你的搜索解决方案时最典型的一种方式，没有用户想要一开始就填写复杂的搜索表单。由于基础关键词搜索是搜索引擎用户最常用的搜索方式，所以理所应当为之提供出色的用户体验。

一般情况下，用户想要输入几个简单的关键词之后得到不错的搜索结果。这听起来像是一项将查询词项与文档进行匹配的简单任务，但为了达到出色的用户体验，必须考虑以下几点：

- 必须快速返回相关结果，大多数情况下应该在一秒之内或更短。
- 在用户拼错一些查询语词时，需要进行拼写纠错。
- 自动建议会减少击键次数，特别对移动应用而言。
- 识别查询词项的同义词。
- 匹配包含查询词项语言形式变化的文档。
- 短语处理是必要的，即用户想要匹配整个短语或部分文字。
- 必须处理查询中包含的“a”、“an”、“the”等常用词。
- 如果用户对排名靠前的搜索结果不满意，必须能继续查看更多结果。

如你所见，如果没有专门的处理方式，这些要求会使得看似基础的功能难以实现。但是对于以 Solr 为代表的搜索引擎，以上几点要求是开箱即用的，使用起来非常方便。为用户提供强大的工具来执行关键词搜索之后，需要考虑的是如何显示搜索结果。这就引出了下一个用例：基于用户查询相关度的搜索结果排名。

#### 排名检索

搜索引擎代表了基于查询返回“优先”文档的一种方法。在关系型数据库的 SQL 查询中，表的一行要么匹配一个查询，要么不匹配，查询结果基于一列或多列进行排序。搜索引擎根据文档与查询匹配的程度为文档赋分，并按降序返回结果。匹配程度的计算取决于多个因素，一般而言，文档得分越高意味着该文档与查询的相关性越强。

根据相关度对文档进行排名是非常重要的，原因如下：

- 现代搜索引擎通常存储大量文档，往往以百万计或数十亿计。若不根据查询相关度对文档进行排名，就会缺乏清晰的导航方式，用户将无法应对搜索结果。
- 用户更习惯于使用那些只需要输入几个关键词就能得到结果的搜索引擎。用户没有耐心，希望搜索引擎能够达到“不用我说，你懂得”的境界。在移动应用背后的搜索中，用户输入存在拼写错误的短查询，并期望其有效，这是现实情况的写照。

通过为特定文档、字段或指定词项赋予更高的权重或加权，可以影响文档排名。例如，通过提升时间字段权重，可以将较新的文档提升到搜索结果的前列。第3章会深入介绍文档排名。

### 超越关键词搜索

使用以 Solr 为代表的搜索引擎，用户可以输入几个关键词并得到结果。但对许多用户来说，这仅仅是交互进程的第一步，他们还将对搜索结果进行不断探索。搜索引擎的一个主要用例是驱动信息发现进程。通常情况下，用户并不准确知道他们要寻找什么，也不清楚搜索系统里有什么信息。一个好的搜索引擎会帮助用户明确他们的信息需求。

此处的核心思想是从初始查询返回文档，以及提供工具帮助用户限定搜索。换句话说，除了返回匹配的文档之外，还应告诉用户下一步该怎么做。例如，根据文档特征对搜索结果进行分类，方便用户缩小搜索范围。这就是所谓的分面搜索，是 Solr 的强项之一。1.2 小节将介绍一个房地产的分面搜索例子，第8章将深入介绍分面。

### 不要将搜索引擎用于……

让我们看一些不适合搜索引擎的用例。首先，搜索引擎会为每个查询返回一小部分文档集合，通常为 10~100 个。同一查询的更多文档可以通过 Solr 内置的分页功能获取。假设一个查询匹配了 100 万个文档，如果一次性全部返回这些文档，等待时间会很长。查询本身的执行速度可能很快，但从底层索引结构上重构 100 万个文档极其缓慢。Solr 使用特定格式将字段存储在磁盘上，这种格式适合于创建少量文档，但生成结果时需要重构大量文档的话，速度会很慢。

另一个用例是，除非内存足够大，否则不要用搜索引擎去完成深度分析任务，这需要访问大部分索引。即使通过分页解决上述问题，底层索引的数据结构也并不适合一次检索大部分索引。

之前已经谈到过这个问题，但这里要重申一下，搜索引擎无法检索文档之间的

关系。Solr 确实支持“父子”关系的查询，但不支持 SQL 复杂关系型数据结构中的关系发现。第 3 章会介绍关系型数据如何适应 Solr 的平面文档结构。

此外，大多数搜索引擎都不直接支持文档级别的安全策略，至少 Solr 没有。如果你需要细粒度的文档权限，这就不属于搜索引擎范畴了。

至此，我们了解了搜索引擎适用和不适用的数据类型和用例，是时候从抽象层面了解 Solr 的工作原理了。下一小节介绍 Solr 能提供哪些功能，以及与外部系统集成、可扩展性与高可用性等重要的软件设计理念。

## 1.2 Solr是什么

本节通过搜索应用设计介绍 Solr 的关键组件。这有助于我们理解 Solr 提供的具体功能及其背后的设计动机。在了解 Solr 到底是什么之前，我们先明确 Solr 不是什么。

- Solr 不是 Google 或 Bing 这样的网络搜索引擎。
- Solr 不具备网站搜索引擎优化（SEO）方面的功能。

假设我们现在需要为潜在的购房者设计一个房地产搜索引擎。这个搜索应用的核心用例是通过网页浏览器搜索出售的房屋。图 1.1 是假想的应用截图。不要太关注 UI 布局设计，这只是为了营造视觉场景而设计的原型，目的是展示 Solr 支持的体验类型。

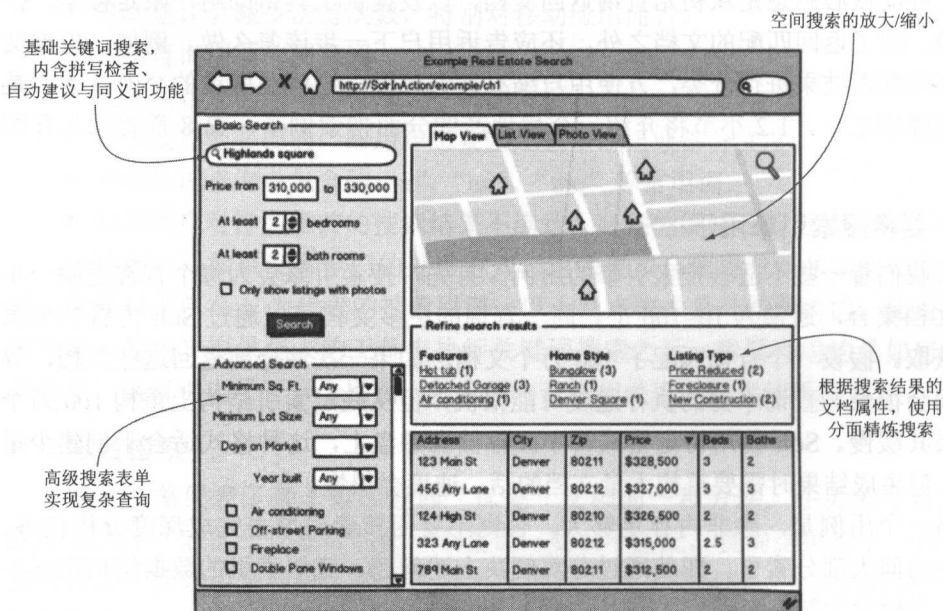


图 1.1 为展示 Solr 功能的虚构搜索应用原型截图

让我们来看看图 1.1 中所示的 Solr 的一些主要功能。我们从左上角开始沿顺时针方向看，Solr 提供了功能强大的关键词搜索框。正如 1.1.2 小节讨论的，基础关键词搜索要做到出色的用户体验，就需要复杂的架构，而 Solr 提供了开箱即用的解决方案。具体而言，Solr 提供了拼写检查（用户输入时提供建议）、同义词处理、短语查询、文本分析等工具，用来处理查询词项的语言变体，如 `buying a house` 或 `purchase a home`。

Solr 还提供了强大的地理空间查询解决方案。如图 1.1 所示，以虚构的邻居为中心，根据经纬度距离，Solr 将匹配的房屋列表显示在地图上。借助 Solr 的地理空间支持，可以根据地理距离对文档排序，在特定范围内过滤文档，甚至根据任意位置返回每个文档的地理距离。地理空间搜索要快速有效，UI 界面要支持缩放和在地图上移动，这些功能很重要。

用户执行查询后，Solr 的分面功能可以对搜索结果进一步细分，显示结果集的文档特征。分面是对结果进行归类的一种方法，用于引导进一步的发现和查询细分。例如，图 1.1 中搜索结果根据功能配套、家居风格和房屋类型三个分面进行归类。

至此，我们对房地产搜索应用的功能类型有了基本认识。下面我们来看看如何在 Solr 中实现这些功能。首先，我们需要知道，Solr 如何对用户输入的查询与索引进行匹配，继而得出房屋列表。这是所有信息检索应用的根基。

### 1.2.1 信息检索引擎

Solr 构建在 Apache 的 Lucene 之上，Lucene 是一个广受欢迎的基于 Java 的开源信息检索库。第 3 章会详细讨论信息检索是什么，这里只介绍信息检索的核心概念。权威的信息检索教科书对现代搜索概念的一个正式定义如下：

信息检索 (Information Retrieval, IR) 是从大规模集合（通常存储在计算机）中查找满足特定信息需求的具有非结构化性质（通常是文本）的资料（通常是文档）的过程。<sup>1</sup>

在房地产搜索应用的示例中，用户的原始需求是根据地理位置、家居风格、功能配套和价格找房子。索引包含全国各地的房屋清单，这绝对算得上一个“大集合”。简而言之，Solr 使用 Lucene 来实现索引文档的核心数据结构，并执行文档搜索。

---

<sup>1</sup> Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze, *Introduction to Information Retrieval* (Cambridge University Press, 2008).



在后台, Lucene 是一个 Java 类库, 用于倒排索引的构建与管理, 倒排索引是专门用于匹配查询词项与文本文档的数据结构。图 1.2 是房地产搜索应用的 Lucene 倒排索引的简化描述。

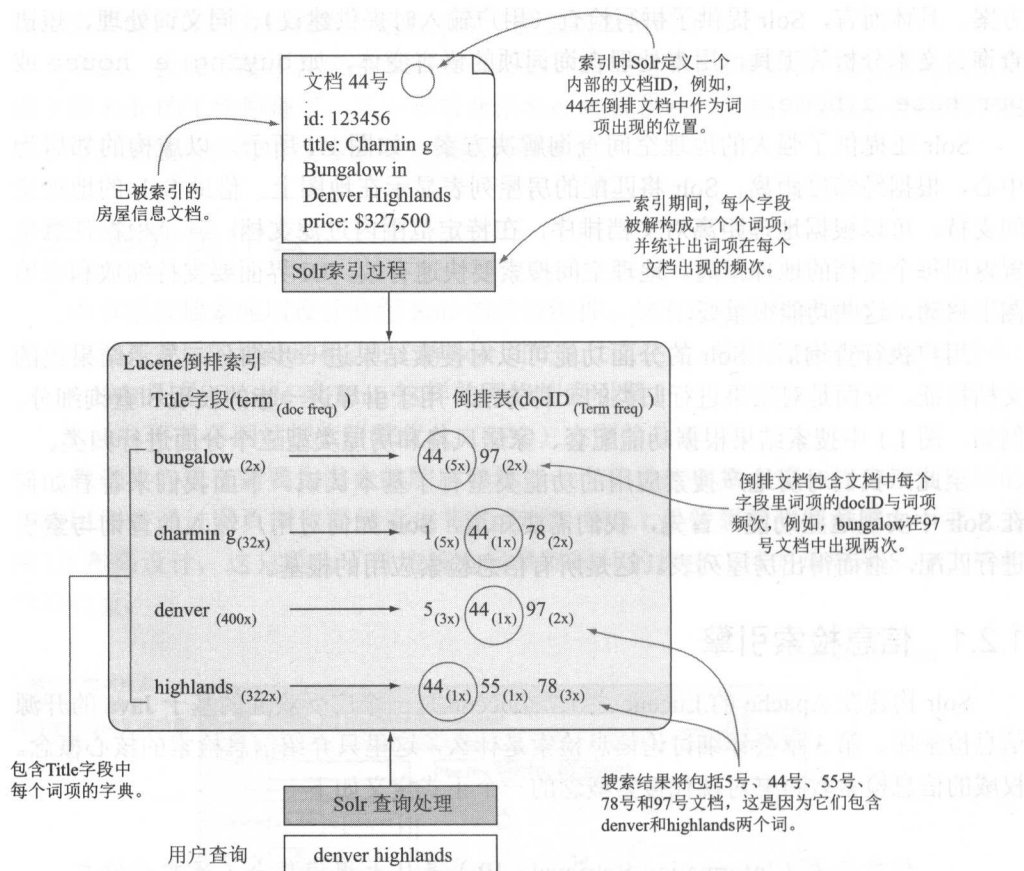


图 1.2 倒排索引是信息检索的关键数据结构

第3章会介绍倒排索引的原理, 这里仅通过图示初步认识倒排索引。在图 1.2 中, 当一个新文档 (44 号) 加入索引后会发生什么, 以及如何使用倒排索引对文档与查询词项进行匹配。

你可能会想, 关系型数据库的 SQL 查询也能很快得到同样的查询结果, 对这个简单的例子而言确实是这样的。但是, Lucene 查询与数据库查询的不同在于, Lucene 的结果是根据查询的相关度进行排名, 数据库的结果只是根据表的一列或多列进行排序。换言之, 文档相关度排名是信息检索有别于其他搜索类型的重要方面。

### 构建 Web 规模的倒排索引

Google 的网页搜索也使用倒排索引，你可能会对此感到吃惊。事实上，正是为了构建 Web 规模的倒排索引才导致了 MapReduce 的诞生。

MapReduce 是将大规模数据处理操作部署在商业服务器集群上的一个编程模型，部署过程包括特定算法实现映射（Map）和规约（Reduce）两个阶段。凭借其函数式编程根基，Google 使用 MapReduce 构建大规模倒排索引，用来驱动网页搜索。在映射阶段会产生唯一词项与文档编号（词项出现的位置）。在规约阶段，词项进行排序，所有的 term/docID 对被发送到同一个规约处理器，规约处理器汇总每个词项的所有词项频次，从而生成倒排索引。

Apache 的 Hadoop 提供了 MapReduce 的开源实现，原属于 Apache 的 Nutch 开源项目，其使用 Solr 为 Web 大规模搜索构建 Lucene 倒排索引。Hadoop 和 Nutch 的深入讨论超出了本书的范围，如果你打算构建大规模索引，建议花时间了解这些项目。

Lucene 提供搜索的核心基础架构，那么 Solr 在 Lucene 基础上做了哪些增强？让我们从如何使用 Solr 的 schema.xml 配置文件来定义索引结构开始了解吧。

### 1.2.2 灵活的模式管理

虽然 Lucene 提供文档索引和查询执行的类库，但缺少生成索引结构的简单配置方法。Lucene 中定义字段和分析字段都需要编写 Java 代码。通过名为 schema.xml<sup>2</sup> 的 XML 配置文件，Solr 提供了一种简单明了的索引结构定义和字段表示及分析方法。Solr 后台使用 schema.xml 表示所有可能的字段和数据类型，将文档映射为 Lucene 索引。这样可以节省编程时间，并让索引结果更易于理解和交流。Solr 构建的索引与 Lucene 编程方式构建的索引是完全兼容的。

Solr 还对 Lucene 核心索引功能进行了扩展。具体来说，Solr 增加了复制字段（copy field）和动态字段（dynamic field）。复制字段取得一个字段或多个字段的原始文本内容，并将其赋予不同的字段。动态字段允许将同一字段类型赋予多个不同字段，而不需要在 schema.xml 中显式声明。这对那些具有多个字段的文档建模非常有用。第 5 章和第 6 章将深入介绍 schema.xml。

在房地产应用的例子中，不用对 schema.xml 做任何修改就可以运行 Solr 示例服务器，你可能会感到惊讶，不过这正显示了 Solr 模式的灵活性。虽然 Solr 示例服务器被设计用于支持产品搜索，但对房地产搜索例子依然有效。

2 译者注 自 Solr 5 之后，Solr 默认使用 managed-schema，通过 Schema API 动态修改模式。schema.xml 是通过直接编辑文件来修改模式。两者的作用是相同的。



Lucene 提供了文档索引、查询执行和结果排名的强大类库。通过 `schema.xml`，你可以使用 XML 配置文档灵活地定义索引结构，无须再用 Lucene 的 API 进行编程。接下来需要一种方法通过网络访问这些服务。下一小节介绍 Solr 如何作为一个 JavaWeb 应用执行，以及如何使用成熟的标准（如 XML、JSON 和 HTTP）与其他技术进行集成。

### 1.2.3 Java Web 应用

Solr 是一个 Java Web 应用，可以运行在任何主流 Java Servlet 引擎中，例如，Jetty 和 Tomcat，或 JBoss、Oracle AS 这样的 J2EE 应用服务器。图 1.3 展示了 Solr 服务器的主要软件构成。

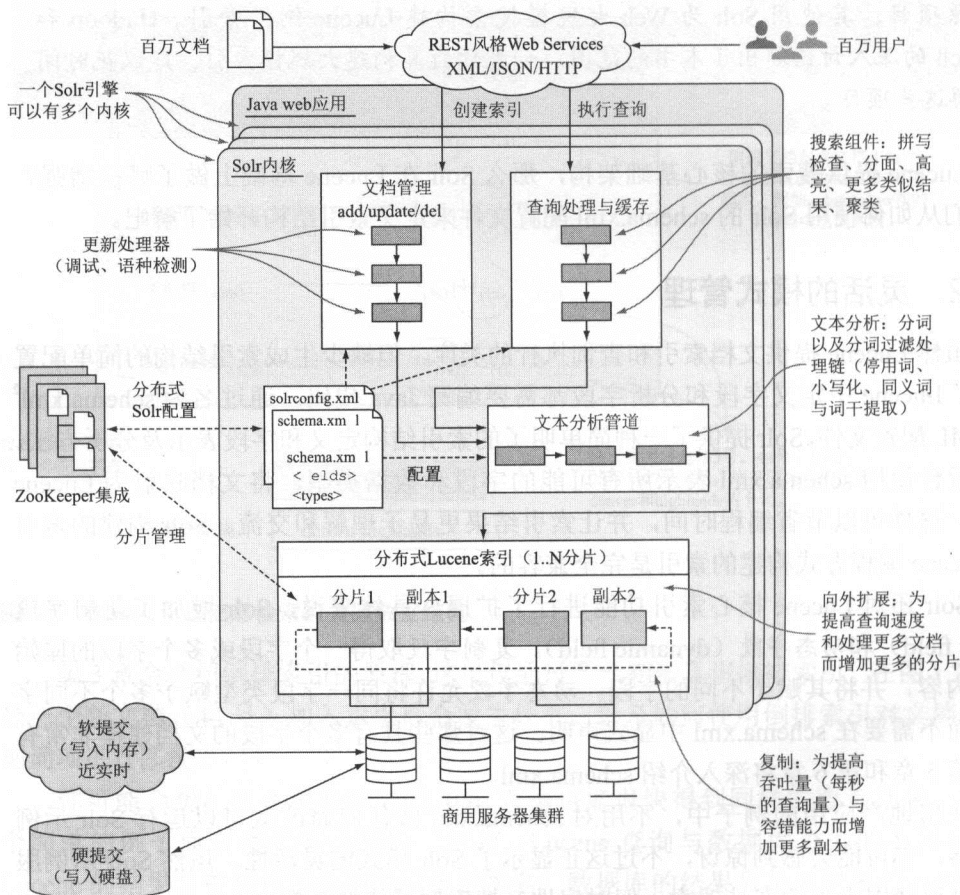


图 1.3 Solr 4 的主要构成图解

显然,图 1.3 初看之下有点复杂。花点时间浏览一下这张图,熟悉一下这些术语。不必担心对其中的术语和概念不了解,读完这本书后,你将对图 1.3 中的概念有深刻理解。

正如本章前面所提及的,Solr 设计者认为,在现有架构下 Solr 是最佳的补充技术。事实上,你很难找到一个 Solr 不适用的环境。在第 2 章你将会看到,下载完几分钟之后你就可以将 Solr 示例服务器运行起来。

为实现轻松集成的目标,Solr 的核心服务需要访问许多不同的应用和编程语言。Solr 基于已有的 XML、JSON 和 HTTP 标准,提供简单的类似 REST 的服务。由于 Solr 并没有严格遵守所有的 REST (Representational State Transfer, 代表性状态传输) 原则,因此要避免为 Solr 基于 HTTP 的 API 打上 REST 风格标记。例如,在 Solr 中,删除文档使用 HTTP 的 POST 方法,而不是用 HTTP 的 DELETE 方法。

将类 REST 接口作为基本原理固然不错,但开发者往往会使用他们喜欢的编程语言的客户端库 (Client Library) 进行访问,以应对 Web Service 调用与响应处理的单一机制 (boilerplate machinery)。所幸的是,大多数编程语言,包括 Python、PHP、Java、.NET 和 Ruby,都有 Solr 的客户端库。

### 1.2.4 一台服务器上的多个索引

现代应用架构的一个标志是应对快速变化要求的灵活性需求。在这样的情况下,Solr 所提供的帮助之一是允许你不必把所有内容放在 Solr 的一个索引里。Solr 支持在单个引擎上运行多个内核。图 1.3 中将多个内核处理成单独的层,全部运行在同一个 Java Web 应用环境下。

每个内核可视为一个单独的索引和配置,并且一个 Solr 实例可以包含多个内核。这样就可以在一台服务器上管理多个内核,共享服务器资源和管理任务,例如,监控和维护。第 12 章将介绍 Solr 进行多内核创建与管理的 API。

Solr 支持多核的一个用途是数据分区,例如,一个内核处理最近的文档,另一个处理较久远的文档,这就是所谓的时间顺序分片 (chronological sharding)。多核的另一个用途是支持多租户应用。

在房地产应用的例子中,多个内核被用于管理不同类型的列表,每个列表都有自己的索引。如果要抛开家用住宅单独来看购买乡下土地的房地产列表,由于在一座城市购买乡下土地与购买房屋的过程不同,因此把土地列表放在单独的核心来管理可能比较合理。

### 1.2.5 可扩展性 (插件)

图 1.3 描述了 Solr 的三个主要子系统:文档管理、查询处理和文本分析。当然,

这都是 Solr 复杂子系统的高度抽象，我们随后会深入介绍每一个子系统。每个子系统都是由模块化的“管道”构成的，通过插件方式实现新功能。这意味着，没有必要推翻 Solr 的整个查询处理引擎，只需要在已有管道中接入新的搜索构件即可。这使得 Solr 的核心功能易于扩展，并能根据特定应用需求实现定制。

### 1.2.6 可伸缩性

Lucene 是一个执行速度相当快的搜索类库，Solr 汲取了 Lucene 速度方面的所有优点。但无论 Lucene 多快，由于 CPU 的 I/O 限制，单台服务器终会达到来自不同用户的并发请求的处理上限。

实现伸缩性的第一步是，Solr 提供灵活的缓存管理功能，帮助服务器重用运算量大的数据扩容。具体来说，Solr 预先设置一些缓存来节省繁重的重新运算量，例如，为查询过滤器缓存搜索结果。第 4 章将介绍 Solr 的缓存管理功能。

缓存能做的事情只有这些，如果需要处理更多的文档和复杂的查询，可以通过增加服务器实现增容。这里，我们关注 Solr 伸缩性的两个最常见维度：查询吞吐量和文档索引量。查询吞吐量是指搜索引擎每秒支持的查询数量。虽然 Lucene 可以快速执行每个查询，但单台服务器上的并发请求存在瓶颈。为实现更高的查询吞吐量，可以通过增加索引副本让更多的服务器处理更多的查询请求。这意味着，如果在三台服务器上复制索引，每秒的查询数量会变为原来的 3 倍，因为每台服务器只用处理三分之一的查询请求。在现实中，完美的线性伸缩性很难实现。因此，增加到三台服务器可能会比一台服务器时增速 2.5 倍。

伸缩性的另一个维度是文档索引量。如果数量规模较大，那么单个实例会因容纳太多文档而达到极限，查询性能也会受影响。为了处理更多文档，可以将索引拆分为很小的索引块，称之为索引分片，然后在索引分片中进行分布式搜索。

#### 使用虚拟化扩展规模

现代计算的一个发展趋势是软件架构的建设能够使用虚拟化商用硬件进行横向扩展。通过添加更多的商用服务器来处理更多的流量。云计算提供商，如 Amazon 的 EC2，助力了这一趋势的发展。Solr 的确可以运行在虚拟化硬件上，不过你应该清楚搜索是 I/O 和内存密集型应用。因此，如果搜索性能对你的组织而言是重中之重，那么就应该考虑在高性能磁盘（如固态硬盘 SSD 非常理想）的高端硬件上部署 Solr。第 12 章将讨论 Solr 部署的硬件知识。

可伸缩性非常重要，故障处理能力对现代系统也很重要。下一小节讨论 Solr 如何处理软件和硬件故障。

### 1.2.7 容错性

除了可伸缩性，还需要考虑一台或多台服务器的故障处理能力，特别是打算在虚拟机或商用硬件上部署 Solr 时。最起码必须有一套故障应对办法。即使是最好的架构和最高端的硬件也会出故障。

假设索引有 4 个分片，托管分片 2 的服务器断电了。这时，Solr 无法继续索引文档和提供查询服务，搜索引擎基本就宕机了。为避免这种情况，需要为每个分片添加副本。当分片 2 发生故障时，Solr 可以启用副本来索引和处理查询，Solr 集群能够保持联机状态。这种情况下故障虽然不会影响索引和查询，但由于少了一个处理请求的服务器，可能会降低速度。第 12 章和第 13 章将讨论故障转移场景。

Solr 拥有先进的、精心设计的架构来满足可伸缩性和容错性。如果想要采用 Solr，这些都需要重点考虑，但仍然不能由此确定 Solr 就是最合适的选择。1.3 节将从不同人员的角度，如软件架构师、系统管理员和 CEO，谈谈采用 Solr 的好处。

## 1.3 选择Solr的理由

本小节会提供一些关键信息，为你的组织决定是否采用 Solr 技术提供决策帮助。首先从软件架构师开始，Solr 对他们很有吸引力。

### 1.3.1 面向软件架构师的 Solr

软件架构师评估一项新技术时，必须考虑许多因素，包括稳定性、可伸缩性与容错性。Solr 在这三方面是遥遥领先的。

在稳定性方面，Solr 是一项成熟的技术，得到活跃的社区和经验丰富的开发者的支持。从开源代码的主工作目录直接部署，无须等待正式版本发布，这一点可能会让 Solr 和 Lucene 的新用户感到吃惊。我们不给具体建议，这里只是为了说明 Lucene 和 Solr 自动化测试的深度与广度。如果是自动化测试全部通过的最新测试版（而非主版本），那么你就完全不用担心其核心功能的稳固性。

1.2.6 和 1.2.7 小节已经介绍了 Solr 的可伸缩性与容错性方式。作为软件架构师，你可能最想了解 Solr 在这两方面的局限性。首先，你应该认识到，Solr 4 中的分片与复制功能被改进了，更加强健和易于管理。新的扩展方式称为 SolrCloud。SolrCloud 后台使用 Apache 的 ZooKeeper 处理 Solr 服务器集群的分布式配置，并跟踪集群的状态。SolrCloud 的一些新功能点如下：

- 集中配置。
- 无单点故障（Single Point of Failure, SPoF）的分布式索引。
- 故障会自动转移到新的分片代表（shard leader）。

- 查询可以发送至集群中的任何一个节点，触发跨所有分片的完整的分布式搜索，内置了故障转移和负载均衡支持功能。

这也不意味着 Solr 的伸缩性没有改进空间。当修改索引大小（合并或分割索引）时，SolrCloud 依然需要人工介入。并不是所有的 Solr 功能都适用于分布式模式。第 12 章将具体介绍 Solr 的伸缩性，第 13 章将专门介绍 SolrCloud 的新功能。希望软件架构师都能了解一点，即在过去几年中 Solr 的伸缩性得到快速发展，且支持无单点故障的稳健伸缩性。

### 1.3.2 面向系统管理员的 Solr

作为系统管理员，采用像 Solr 这样的新技术时，需要优先考虑新技术是否适合现有的基础架构。简单的答案是：Solr 当然适合。因为 Solr 是基于 Java 的，可以运行在任何具有 J2SE 6.x/7.x JVM 的操作系统平台上。Solr 内置了一个由 Oracle 提供的开源 Java Servlet 引擎 Jetty。另外，Solr 是标准的 Java Web 应用，很容易部署在 Java Web 应用服务器中，如 JBoss 和 Apache 的 Tomcat。

Solr 的所有访问都要通过 HTTP，支持 HTTP 缓存反向代理 Squid 和 Varnish。Solr 还支持 JMX，能够挂载监控应用，如 Nagios。

另外，Solr 还提供一个很不错的管理控制台，检查配置设定、查看统计数据、提交测试查询和监控 SolrCloud 的健康状况。图 1.4 是 Solr 的管理控制台界面，第 2 章会具体介绍。

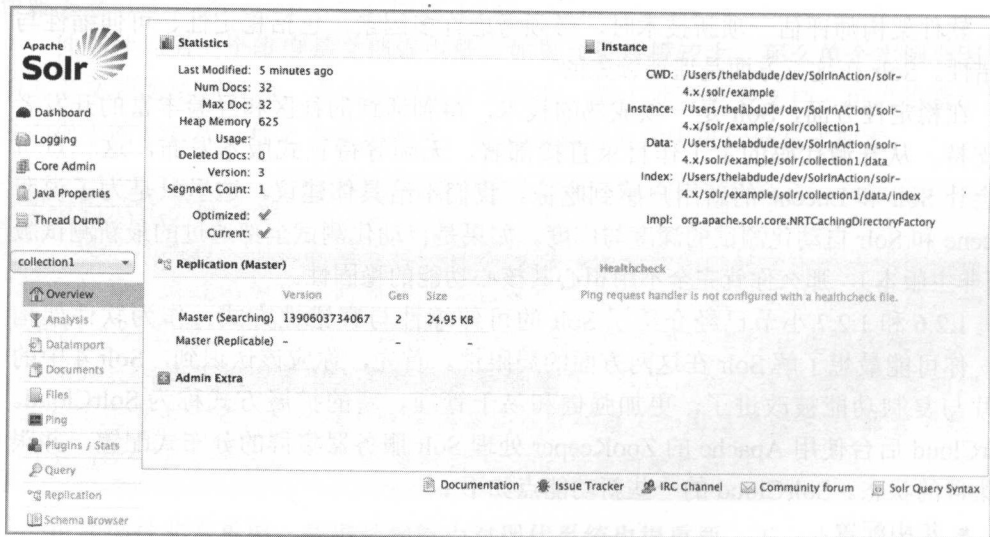


图 1.4 Solr 4 的管理控制台截图，在其中可以发送测试查询、检查服务器网络、查看配置设定，以及掌握集群中分片与副本的分布情况



### 1.3.3 面向 CEO 的 Solr

尽管很多 CEO 不会阅读这本书，但我们还是在这里提供一些与 CEO 当面讨论搜索的谈资。

- 高管想要知道当今投资一项技术的长期回报。你可以强调许多公司仍然还在运行 2009 年发布的 Solr 1.4，这意味着 Solr 是成功的，并在不断完善。
- CEO 们喜欢可预见的技术。下一章你就会看到，Solr 如此有效，在几分钟内就能启动与运行起来。
- Solr 拥有庞大的社区支持。如果一个 Solr 技术人员外出会发生什么，业务会中止吗？Solr 确实是一项复杂的技术，但它背后有活跃的社区支持。也就是说，需要时会有援手。你也可以访问 Solr 的源代码，一旦出了什么问题需要修复，自己也可以动手处理。许多商业服务商也提供 Solr 的规划、实施和维护，还提供 Solr 的培训课程。
- Solr 入门不需要太多初始投资（可能 CFO 对这一点更感兴趣）。无论环境的大小和规模如何，我们都有信心保证你在几分钟之内就启动一个 Solr 服务器，并快速索引文档。运行在云中一台普通服务器能够处理百万级数量的文档，以亚秒级速度响应众多查询请求。

## 1.4 功能概述

最后，让我们从以下三个方面快速了解一下 Solr 的主要功能：

- 用户体验功能
- 数据建模功能
- Solr 4 的新功能

搜索解决方案提供出色的用户体验是贯穿本书的重要理念，因此让我们先了解一下 Solr 是如何让用户感到愉悦的。

### 1.4.1 用户体验功能

Solr 提供一些重要功能，使得搜索解决方案简单易用、直观且功能强大。应该注意一点，Solr 只提供类 REST 的 HTTP API，不为其他编程语言或框架提供搜索相关的 UI 组件。你可以借鉴下面的用户体验功能，自己动手开发一些搜索 UI 组件：

- 分页与排序
- 分面
- 自动建议
- 拼写检查

- 搜索结果高亮
- 地理空间搜索

## 分页与排序

Solr 支持分页请求，不用一次性返回所有匹配的文档，第一页只返回前  $N$  个文档，如果用户在第一页没有找到想要的，可以使用简单的 API 请求参数请求后续页面。分页有两个重要作用：(1) 更迅速地返回结果，因为每个请求只返回了整个搜索结果的一小部分；(2) 有助于跟踪哪些查询产生了多个页面请求，这可能会为相关度评分问题提供参考。第 7 章会详细介绍分页与排序。

## 分面

分面对搜索结果进行归类分组，帮助用户限定查询条件和发现更多信息。在房地产例子中（参见图 1.1），基础关键词搜索的搜索结果被归类到功能配套、家居风格、房屋类型等三个分面中。分面是 Solr 最受欢迎的强大功能之一。第 8 章会深入介绍分面。

## 自动建议

大多数用户会期望搜索应用“做正确的事”，即便自己只提供了不完整的信息。自动建议能够为用户提供一个建议列表，包含基于索引文档的词条和短语。Solr 的自动建议功能是在用户开始输入几个字符时，根据输入情况显示一个建议查询列表。自动建议能够减少错误查询的数量，特别是在许多用户使用移动设备的小键盘进行搜索的情况下。

自动建议向用户展示索引中可用的词条与短语。在房地产例子中，当用户输入 `hig` 时，Solr 的自动建议功能会给出类似 `highlands neighborhood` 或 `highlands ranch` 的建议。第 10 章会详细介绍自动建议。

## 拼写检查

在移动设备满天下且人们都忙忙碌碌的年代，拼写纠错的支持必不可少。再次强调，用户期望搜索引擎能妥善处理拼写错误。Solr 的拼写检查支持以下两种基本模式：

- 自动更正——如果索引中存在拼写错误的词条，Solr 会自动进行更正。
  - 你要找的是不是——Solr 给出的建议查询可能会得到更好的结果，因此当用户输入 `hilands` 时可以给出一条提示，例如“你要找的是不是 `highlands`”。
- Solr 4 的拼写纠正经过改进后更易管理和维护，第 10 章将介绍其工作原理。

## 搜索结果高亮

当搜索大量文本文档时，可以使用 Solr 的搜索结果高亮功能显示出结果文档的

特定部分。搜索结果高亮对越长的文档效果越好。对与用户查询匹配的搜索结果的特定部分进行高亮显示，有助于用户找到相关文档。特定的文本部分基于查询相似度动态生成。第 9 章将介绍搜索结果高亮。

### 地理空间搜索

地理位置是 Solr 4 的高级功能，内置了经度值和纬度值的索引支持，并根据位置距离对文档排序。Solr 可以根据地理位置（经度和纬度）对文档进行查找和排序。在房地产例子中，搜索匹配的结果显示在一张交互地图上。用户使用地理空间搜索，通过将地图放大和缩小、移动地图中心点来找到附近的房屋。

Solr 4 另一个令人兴奋的附加功能是支持地理形状的索引，如多边形，用以寻找位置相交的地理区域。居民区用精确的地理位置来表征，有助于寻找特定居民区的房源。第 15 章介绍地理空间搜索

## 1.4.2 数据建模功能

1.1 小节讨论了 Solr 支持的数据类型。本节简要介绍搜索的数据建模功能，具体内容如下：

- 结果分组 / 字段折叠
- 灵活的查询支持
- 连接
- 文档聚类
- 导入各种文档格式，如 PDF 和 Word
- 从关系型数据库中导入数据
- 多语种支持

### 结果分组 / 字段折叠

虽然 Solr 要求平面、非规范化文档，但我们可以根据文档的共同属性将多个文档归为一组。结果分组，也称字段折叠，返回的结果是成组的，而不是单个文档。

字段折叠的典型例子是线程化的电子邮件讨论。根据特定的查询，邮件会在发起对话的第一个邮件消息下面被分组。第 11 章介绍结果分组 / 字段折叠。

### 灵活的查询支持

Solr 提供许多强大的查询功能，包括：

- “与” (AND)、“或” (OR)、“非” (NOT) 条件逻辑
- 通配符匹配
- 支持日期和数字的区间查询



- 支持词项间隔距离的短语查询
- 模糊字符串匹配
- 正则表达式匹配
- 函数查询

第7章将详细介绍这些查询功能。

## 连接

SQL 的连接 (Join) 使用两个及以上的表之间的共同属性 (如外键) 来抽取数据, 从而创建关系, Solr 的连接更像是 SQL 的子查询, 不用连接其他文档的数据来创建文档。Solr 的连接返回符合搜索条件的子文档。使用单一响应返回一条推文的所有转发, 这是 Solr 的一种连接应用。第15章将讨论连接。

## 文档聚类

文档聚类根据代表每个文档的词项, 将相似的文档分在一组。聚类有助于避免搜索结果中返回很多包含相同信息的文档。例如, 要从多个 RSS 供稿中获取新闻文章的搜索引擎, 很有可能得到同一条新闻的多个文档。我们可以通过聚类挑选一条具有代表性的文章, 而不是返回同一条新闻的多个文档。第16章会简要讨论聚类技术。

## 导入各种文档格式, 如 PDF 和 Word

某些情况下需要对 PDF 或 Word 等常见格式的大量文档进行搜索。对 Solr 来说这很容易做到, 因为它集成了 Apache 的 Tika 项目, 而 Tika 支持大多数流行的文档格式。第12章介绍各种文档格式的导入。

## 从关系型数据库中导入数据

要用 Solr 搜索关系型数据库中的数据, 通过配置 Solr 使用 SQL 查询生成文档。第12章将介绍 Solr 的数据导入处理器 (DIH)。

## 多语种支持

Solr 与 Lucene 有很长的多语种处理历史。Solr 内置语种检测, 并提供多语种的特定文本分析解决方案。第14章介绍 Solr 的语种检测与多语种文本分析。

### 1.4.3 Solr 4 的新功能

在本章结束之前, 让我们看看 Solr 4 令人兴奋的新功能。对 Apache 的 Solr 社区而言, Solr 4 是一个重大的里程碑。它解决了过去几年里真实用户发现的许多重要问题。这里介绍一些主要功能, 而 Solr 4 的新功能介绍会贯穿全书。

- 近实时搜索
- 原子更新与乐观并发
- 实时 GET 功能
- 使用事务日志实现写持续性
- 使用 ZooKeeper 实现简易分片和复制

### 近实时搜索

Solr 的近实时搜索 (Near Real-Time, NRT) 功能实现了文档添加到索引后的几秒钟之内, 就能很快被搜索到。借助近实时搜索, Solr 可以搜索快速变化的内容源, 如最新报道和社交网络。第 13 章介绍近实时搜索技术。

### 原子更新与乐观并发

原子更新功能允许客户端应用在已有文档上添加、更新、删除和对字段增值, 而且无须重新发送整个文档。例如, 对于 1.2 小节中的房地产应用, 假设房屋价格变化了, 向 Solr 发送一次专门更改价格字段的原子更新即可。

你可能会问, 如果两个用户同时更改同一个文档会怎样? Solr 使用乐观并发机制提防不兼容的更新。简单来说, Solr 使用特殊的 `_version_` 版本字段来确保文档的安全更新语义。对于两个用户同时更改同一个文档的情况, 后提交更改的用户将会获得一个过时版本字段, 所以会更新失败。第 5 章会介绍原子更新与乐观并发。

### 实时 GET 功能

本章开头说过 Solr 是一种 NoSQL 技术, Solr 的实时 GET 功能是典型的 NoSQL 方式。无论文档是否提交到索引, 你都可以使用唯一标识符检索最新版本。这与使用行键 (row key) 检索数据的 Cassandra 的键-值存储方式类似。

在 Solr 4 出现之前, 除非文档提交到 Lucene 的索引, 否则是检索不出来的。借助 Solr 4 的实时 GET 功能, 我们就不必非要提交后通过唯一 ID 检索文档了。在发送给 Solr 之后且提交之前, 实时 GET 功能对更新已有文档非常有用。我们在第 5 章会了解到, 提交很花时间, 而且会影响查询性能。

### 使用事务日志实现写持续性

当文档发送到 Solr 进行索引时, 会被写入事务日志中, 以防止服务器发生故障造成数据丢失。Solr 的事务日志处在客户端应用与 Lucene 索引之间, 也对实时 GET 请求起作用, 无论是否已经提交给 Lucene, 都可通过唯一标识符检索到文档。

Solr 的事务日志解除了更新可见性与更新持久性的绑定。这就是说, 文档可以持久存储, 但不出现在搜索结果中。事务日志可以控制提交的文档在搜索结果中出现的时机, 避免在提交之前服务器出故障时丢失数据。第 5 章将深入讨论持久性写入与提交策略。

## 使用 ZooKeeper 实现简易分片和复制

如果你是 Solr 新手，可能不会注意到先前版本的扩容是烦琐的手工操作这一事实。SolrCloud 让扩容变得简单和自动化，因为 Solr 使用 Apache 的 ZooKeeper 对分片代表和副本进行配置管理。Apache 网站 (<http://zookeeper.apache.org>) 这样介绍 ZooKeeper: “一套维护配置信息、命名、提供分布式同步和分组服务的集中化服务。”

在 Solr 中，ZooKeeper 负责指定分片代表与分片副本，并对服务请求可用的服务器进行跟踪。SolrCloud 与 ZooKeeper 是捆绑的，因此 SolrCloud 的启动无须做任何额外的配置和安装。第 13 章将详细介绍 SolrCloud。

## 1.5 本章小结

希望你已经对 Solr 支持的数据类型和用例有了很好的认识。1.1 小节提到，Solr 处理的数据是以文本为中心、读主导、面向文档的，并且具有灵活的模式。本章还指出以 Solr 为代表的搜索引擎不是通用的数据存储与处理解决方案，而是提供强大的关键词搜索、排名检索与信息发现。以虚构的房地产搜索应用为例，我们了解了 Solr 如何基于 Lucene 添加声明式索引配置和提供基于 HTTP、XML 和 JSON 的 Web 服务。Solr 4 通过分片与复制两个维度进行扩展，支持百万级数量的文档与高流量的查询请求。Solr 4 在分布式 SolrCloud 配置下不支持 SPoF。

本章还从不同人员的角度讨论了选择 Solr 的理由，谈论了软件架构师、系统管理员，甚至还有 CEO 对 Solr 的关注点。最后介绍了 Solr 的一些主要功能，给出了每个功能介绍对应的后续章节。

希望你能保持兴趣继续学习 Solr。现在是时候下载 Solr 并在本地系统中运行它了，让我们一起进入第 2 章。

# Solr上手

## 本章要点

- 下载并安装 Apache Solr 4.7
- 运行 Solr 的示例服务器
- 实现排序、分页和结果的格式化
- 探索 Solr 的示例搜索用户界面

刚开始接触一门陌生的技术时，感到不安是很正常的，但你可以对 Solr 放心，因为它非常易于安装和使用。为了快速上手，你可以从 Solr 的基础入手，逐步增加 Solr 配置的复杂度。例如，Solr 可以进行分片处理——把一个大的索引拆分为小的子集，通过增加副本来增强查询服务能力。在遇到扩容难题之前，你先不必担心有关索引分片或副本问题。

学完本章，你将能够在自己的计算机上运行 Solr，知道如何启动和停止 Solr，掌握 Web 管理控制台的使用方法，并对 Solr 主目录、内核、集合等核心概念所有了解。

## SolrCloud 的解释及其与 Solr 4 的区别

你可能已经听说过 SolrCloud，也想了解 Solr 4 与 SolrCloud 的区别。从技术层面理解，SolrCloud 是 Solr 4 功能子集的代号，让 Solr 服务器易于配置并运行在可扩展的、具有容错能力的集群上。SolrCloud 可视为 Solr 4 分布式安装的一种配置方法。

此外，虽然 Solr 可以在类似 Amazon 的 EC2 这样的云计算环境中运行，但 SolrCloud 与此并无干系。我们推测，SolrCloud 中的“Cloud”反映了其潜在目标——期望借助云服务来促进灵活可扩展性、高可用性与易用性。第 13 章会深入介绍 SolrCloud。

让我们从 Apache 官方网站下载 Solr 并把它安装在本机上，开启学习之旅吧。

## 2.1 开始上手

学习 Solr 之前，必须先在本地上计算机上运行它。首先，请从 Apache 官方网站下载 Solr 4.7 的二进制分发版，然后解压缩文件。安装完成后，我们将向你展示如何启动 Solr 的示例服务器，并通过浏览器访问 Solr 的管理控制台来确认 Solr 是否运行起来了。要顺利完成这个过程，我们假定你能在你的操作系统中执行简单的命令行语句。Solr 没有图形化用户界面的安装程序，不过你很快会发现并不需要，因为安装非常简单。

### 2.1.1 Solr 的安装

用“安装 Solr”来形容整个过程或许有点用词不当，因为需要做的仅仅是下载二进制分发版（.zip 或 .tgz 格式），并对其解压缩。开始之前，请确保已安装了 Java 1.6 或更高版本（又称 J2SE 6）。要确认是否已安装了正确的 Java 版本，请打开命令行窗口，输入以下命令：

```
java -version
```

你应该看到以下类似的内容输出：

```
java version "1.6.0_24"  
Java™ SE Runtime Environment (build 1.6.0_24-b07)  
Java HotSpot™ 64-Bit Server VM (build 19.1-b02, mixed mode)
```

如果还没有安装 Java，建议使用 Oracle 公司的 JVM（[www.oracle.com/technetwork/java/javase/downloads/index.html](http://www.oracle.com/technetwork/java/javase/downloads/index.html)）。虽然 Solr 的服务器端要求使用 Java，但这也并不意味着你必须在应用程序中通过 Java 与 Solr 进行交互。客户端与 Solr 通过 HTTP 进行交互，所以你可以使用任何提供 HTTP 客户端库的编程语言。此外，.NET、Python、Ruby、PHP 及 Java 等主流编程语言都有可用的 Solr 开源客户端库。

如果 Java 已经安装好，接下来就可以安装 Solr 了。Apache 官方网站提供两种下载方式：Solr 的源代码和二进制分发版。这里重点介绍二进制分发版的安装步骤。

第 12 章会介绍如何从源代码编译 Solr。

要下载 Solr 的最新版本，请前往站点 <http://lucene.apache.org/solr>，然后点击右侧 Apache Solr 的下载按钮，你将进入一个提供 Apache 下载的镜像站点（从镜像站点下载最新版本是个明智的选择，可以避免 Apache 主站点访问超载）。如果你使用的是 Windows 操作系统，请下载 solr-4.7.0.zip。如果使用的是 UNIX、Linux 或 Mac OS X 操作系统，请下载 solr-4.7.0.tgz。本书中的示例基于 Solr 4.7.0 版本，如果 Apache 主页不提供 Solr 4.7.0 版本的下载，而你又想操作书中的示例，可以在 Apache 软件基金会存档库（<http://archive.apache.org/dist/lucene/solr/4.7.0/>）中找到 Solr 4.7.0。

将下载后的文件妥善保存到本地计算机上，若使用 Windows 系统则将其移动到 C 盘根目录下，若使用 Linux 系统则选择类似 /opt/solr/ 的路径。对于 Windows 用户，强烈建议把 Solr 解压缩到名称不含空格的目录中，也就是说，避免出现将 Solr 解压缩到这样的目录下：C:\Documents and Settings\ 或 C:\Program Files\，由于 Solr 是 Java 应用程序，所以安装路径中包含空格很可能会遇到问题。

Solr 是独立的自包含 JAR 归档文件，它不需要专门的安装程序，只需解压缩即可。解压后的所有文件位于目录 solr-4.7.0/ 之下。在 Windows 系统中可以使用内置的 ZIP 解压工具，如 WinZip。在 UNIX、Linux 或 Mac 系统中，请输入 `tar xzf solr-4.7.0.tgz` 的指令，创建如图 2.1 所示的目录结构。

后续章节将解压后 Solr 文件（.zip 或 .tgz 格式）的安装位置统一称为 `$SOLR_INSTALL/`，使用这个名称是因为它看上去较为简洁。由于每个使用者的 Solr 的根目录可能都不相同，所以我们不想使用 `$ SOLR_HOME` 作为解压缩 Solr 的顶级目录的名称。现在，Solr 已经安装完毕，可以启动它了。

## 2.1.2 启动 Solr 的示例服务器

要启动 Solr，请打开命令行窗口，输入以下命令：<sup>1</sup>

```
cd $SOLR_INSTALL/example
java -jar start.jar
```

请记住，`$SOLR_INSTALL/` 表示 Solr 解压缩的目录别名，例如，Windows 上的 `C:\solr-4.7.0\`。现在让我们启动 Solr。

1 译者注 在 Solr 6 系列版本中，Solr 示例服务器启动命令有所变化。在 `$SOLR_INSTALL` 目录下，输入以下命令：

```
bin\solr.cmd start -e techproducts (UNIX、Linux、Mac 系统)
```

```
bin\solr start -e techproducts (Windows 系统)
```

另外，该命令会连同 2.1.4 小节提到的示例文档一并提交生成索引。

初始化过程中，控制台会显示一些日志消息。如果一切顺利，接近日志底部的地方会显示如下消息：

```
3504 [main] INFO org.eclipse.jetty.server.AbstractConnector - Started
SocketConnector@0.0.0.0:8983
```

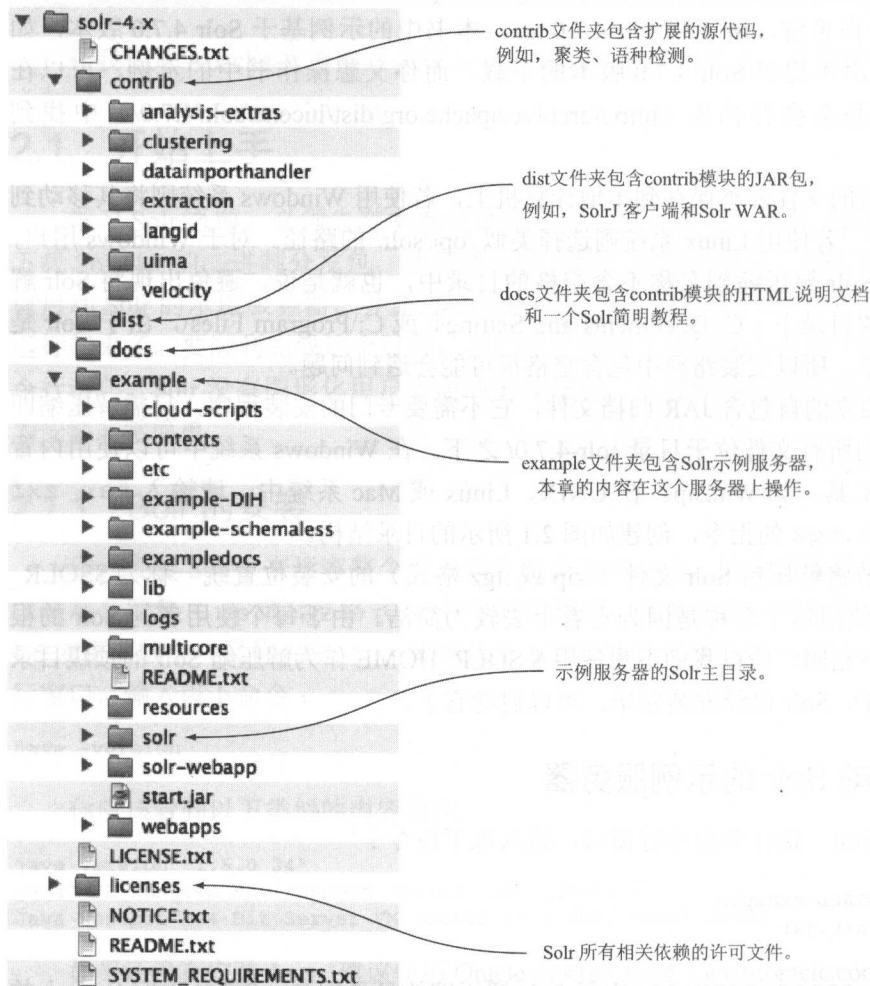


图 2.1 Solr 4.7.0 解压缩后的目录列表。后续章节中会将解压后的 Solr 文件顶级目录(.zip 或 .tgz 格式)称为 \$ SOLR\_INSTALL/

### 到底发生了什么

这似乎太简单了，你可能想知道刚才的命令执行后完成了什么工作。明确说，你的计算机正在运行着 Solr 4.7。通过浏览器访问 <http://localhost:8983/solr>，观察能



否跳转到 Solr 管理页面以确认 Solr 是否正确启动了, 图 2.2 是 Solr 管理控制台的截图, 先来快速熟悉一下控制台的布局和导航工具。

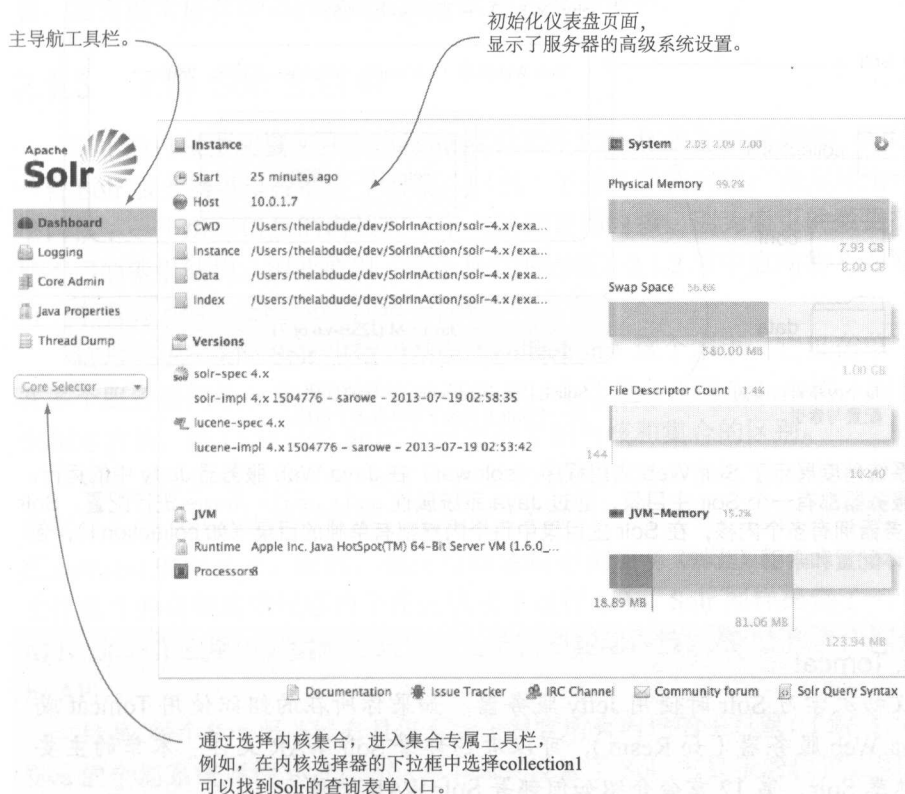


图 2.2 Solr 4.7 管理控制台为新的 Solr 实例运行提供了丰富的工具。点击 collection1 的链接会看到更多工具, 包括查询表单

在后台, start.jar 启动了一个名为 Jetty 的 Java Web 服务器, 监听端口为 8983。Solr 是运行在 Jetty 上的 Web 应用程序。图 2.3 展示了 Solr 的运行情况。

### 故障排除

启动示例服务器应该不会出现太多错误。服务器没有正常启动的最常见原因是默认端口 8983 已经被其他进程占用。如果出现这种情况, 你会看到类似 java.net.Bind -Exception:Address already in use 的错误提示。通过改变 Solr 的绑定端口可以很容易地解决这个问题。命令 `java -Djetty.port=8080 -jar start.jar` 为 Jetty 指定一个不同的端口。使用此命令后, Jetty 将由原先的 8983 端口变为 8080 端口。



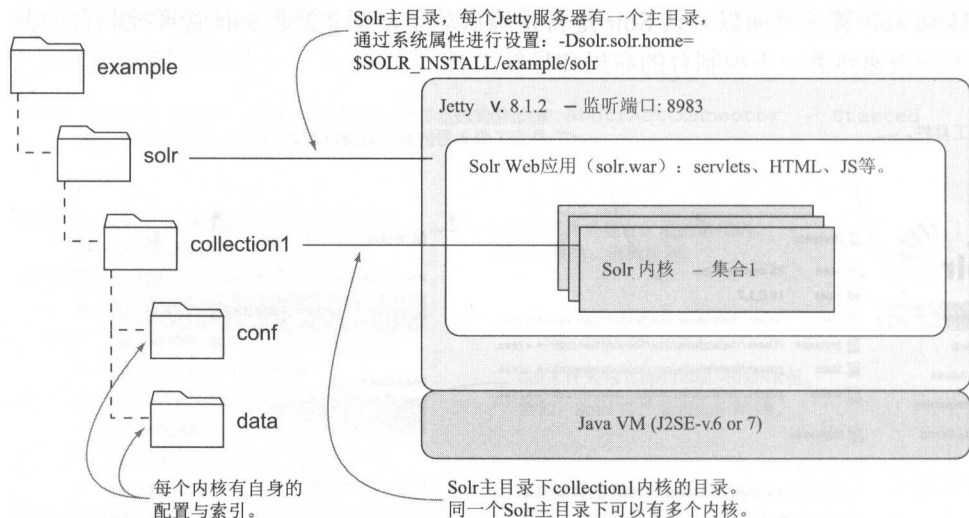


图 2.3 从系统角度展示了 Solr Web 应用程序 (solr.war) 在 Java Web 服务器 Jetty 中的运行。每个 Jetty 服务器都有一个 Solr 主目录，通过 Java 系统属性 `solr.solr.home` 进行配置。Solr 允许一台服务器拥有多个内核，在 Solr 主目录中每个内核都有单独的目录（如 `collection1`），包含内核的具体配置和索引（数据）

### Jetty vs. Tomcat

建议初次学习 Solr 时使用 Jetty 服务器。如果你所在的组织使用 Tomcat 或其他 Java Web 服务器（如 Resin），可以考虑部署 Solr WAR 文件。本章的主要目的是熟悉 Solr，第 12 章会介绍如何部署 Solr WAR 文件。

Solr 在 Jetty 中的初始化安装和配置过程是非常简单的，但这并不意味着 Jetty 不适合作为 Solr 生产环境的部署。如果你的组织已经拥有一个标准的 Java Web 应用平台，那么 Solr 在该平台运行没有问题。如果你可以选择应用平台，建议使用 Jetty。Jetty 速度快、稳定、成熟，并且易于管理和定制。Google 在其 App Engine 上使用 Jetty，参见 <http://www.infoq.com/news/2009/08/google-chose-jetty/>，这一事实表明，将 Solr 运行在 Jetty 上，即使在最苛刻环境下，Jetty 可靠的平台表现同样会令人充满信心。

### 停止 Solr

对于本地操作，可以在启动 Solr 的控制台窗口中按 `Ctrl + C` 组合键来停止 Solr 服务器。通常来说，这种方式对于开发和测试而言已经足够安全。而 Jetty 的确也提供一种更安全的服务器停止机制，这将在第 12 章中讨论。

现在，我们有一个正在运行的服务器，接下来花点时间来了解 Solr 从何处得到自身的配置信息以及如何管理 Lucene 索引。了解已经启动的示例服务器如何进行配置，这有助于你在自己的应用程序中配置 Solr 服务器。

### 2.1.3 了解 Solr 主目录

Solr 的内核由配置文件、Lucene 索引文件和 Solr 事务日志组成。Jetty 上运行的一台 Solr 服务器可以控制多个内核。回想一下第 1 章的房地产搜索应用，它拥有两个内核：一个房屋信息的内核和一个土地列表的内核。因为索引的数据拥有两种截然不同的索引结构，因此使用了两个独立的内核。2.1.2 节中启动的 Solr 示例服务器有一个名为 `collection1` 的单一内核。

这里多说一句，Solr 也使用集合（collection）这个术语，它仅在单一索引分布在多个服务器的 Solr 集群语境下才有意义。此处重点介绍 Solr 内核，因为它理解起来相对容易。第 13 章讨论 SolrCloud 时会介绍内核和集合的区别。

Solr 主目录是封装一个或多个内核的目录结构，在以前版本中，Solr 主目录由配置文件 `solr.xml` 进行配置。从 Solr 4.4 以后，内核可以被自动发现，不再需要在 `solr.xml` 中定义了。因此，现在可以忽略示例服务器提供的 `solr.xml` 文件。这个文件包含的高级选项仅适用于在云模式下运行 Solr。Solr 同样提供了一个内核管理 API，允许在应用中以编程方式创建、更新和删除内核。第 12 章会详细介绍内核管理 API。

目前，每个 Solr 服务器有且仅有一个包含所有内核的主目录，了解这点非常重要。Java 的全局系统属性 `solr.solr.home` 设置了 Solr 主目录的位置。图 2.4 显示了示例服务器中默认的 Solr 主目录——`solr`。

第 4 章介绍 Solr 内核的主要配置文件 `solrconfig.xml`。此外，`schema.xml` 也是主要的配置文件，用于管理文档和查询的索引结构与文本分析。第 5 章会详细介绍 `schema.xml`。现在，让我们通过图 2.4 对 Solr 主目录的结构有一个基本认识。

该示例目录包含另外两个 Solr 的主目录，用于探索高级功能。具体来说，Solr 的 `example/example-DIH/` 目录提供一个有关 DIH 功能的 Solr 内核。`example/multicore/` 目录提供了多内核配置示例。书中稍后会进一步讲述这些功能。现在，我们来练习如何将一些文档添加到索引中，这些会在 2.2 节中用到。

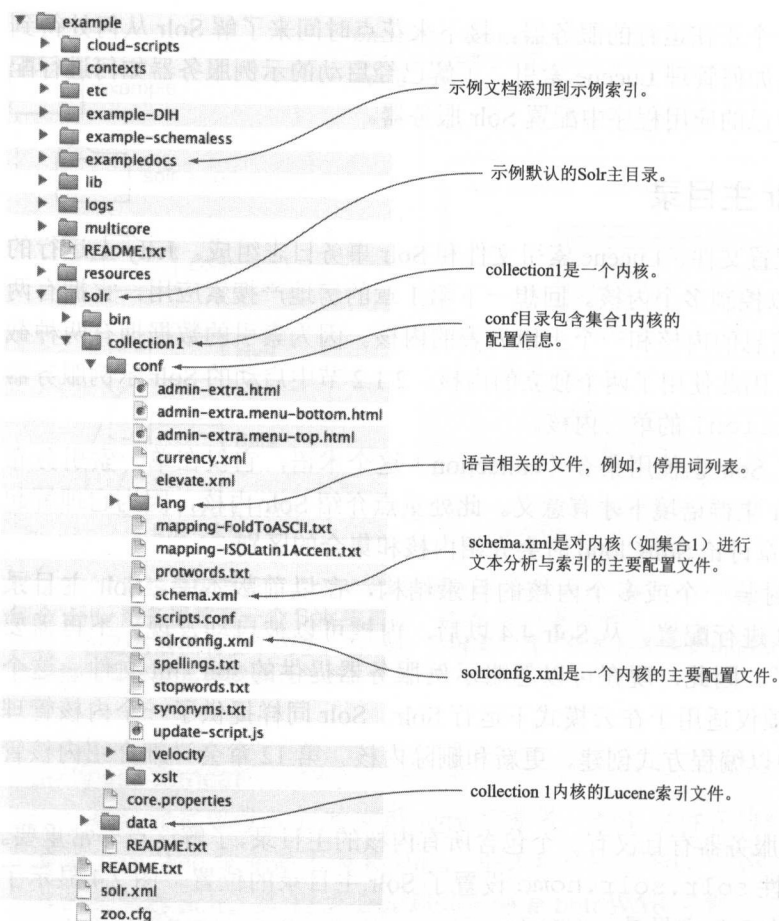


图 2.4 Solr 示例服务器的默认 Solr 主目录。它包括一个名为 collection1 的单一内核，通过 solr.xml 进行配置。collection1 的目录对应 collection1 内核，包括具体的内核配置文件、Lucene 索引及事务日志

## 2.1.4 对示例文档进行索引

第一次启动 Solr，索引中并没有文档。也就是说，这是一台空服务器，等待数据填充以供搜索使用，第 5 章会详细介绍索引构建，这里暂时忽略细节，只是将示例数据装入 Solr 的索引，以便尝试提交查询。打开一个新的命令行窗口，输入以下命令：

```
cd $SOLR_INSTALL/example/exampledocs
java -jar post.jar *.xml
```

应该会看到类似下面的输出：

```
SimplePostTool version 1.5
Posting files to base url http://localhost:8983/solr/update using content-
type application/xml..
POSTing file gb18030-example.xml
POSTing file hd.xml
POSTing file ipod_other.xml
POSTing file ipod_video.xml
POSTing file manufacturers.xml
POSTing file mem.xml
POSTing file money.xml
POSTing file monitor.xml
POSTing file monitor2.xml
POSTing file mp500.xml
POSTing file sd500.xml
POSTing file solr.xml
POSTing file utf8-example.xml
POSTing file vidcard.xml
14 files indexed.
COMMITting Solr index changes to http://localhost:8983/solr/update..
```

post.jar 文件通过 HTTP POST 方式把 XML 文档发送至 Solr。在所有文档被发送至 Solr 后, post.jar 会发起一个提交, 确保示例文档在 Solr 中可被搜索到。在 Solr 管理控制台中前往查询页面 (<http://localhost:8983/solr>), 执行查找所有文档的查询 (\*:\*), 以确认示例文档是否已经添加成功。通过左侧下拉框选择 collection1, 进入查询页面。图 2.5 展示了执行“查找所有文档”后应该看见的结果。

此时, 我们就有了一个加载了一些示例文档的 Solr 运行实例。

## 2.2 一切都关乎搜索

现在是时候见识 Solr 的强大之处了。毋庸置疑, Solr 最强大功能的是查询处理。试想一下, 如果搜索引擎返回的结果不适用或不准确, 谁还会在意它的可扩展性和速度? 本节介绍 Solr 的查询处理表现, 让你见识一下 Solr 强大的搜索技术。

在整个小节中, 请密切关注所执行的每一次查询与 Solr 返回文档之间的联系, 特别是搜索结果中文档的次序。这有助于你思考搜索引擎的工作方式, 接下来的第 3 章将介绍搜索的核心概念。

### 2.2.1 Solr 查询表单详解

前面已经使用过 Solr 查询表单来执行“查找所有文档”。现在快速了解一下查询表单的其他功能, 从而对 Solr 支持的查询类型有所认识。图 2.6 对表单的关键区域做了一些注释, 请花费一点时间阅读图表中的每一个注释。

在图 2.6 中, 我们构造了一条查询, 返回两条示例文档结果 (2.1.4 节曾添加过)。在你自己的服务器上练习填写该表单, 并执行该查询, 查看 Solr 返回的两个文档是

否有意义。表 2.1 介绍了该查询中所使用的表单字段。

在Solr中找到所有文档的查询表达式为\*:\*

找到所有文档的查询执行后得到的搜索结果。

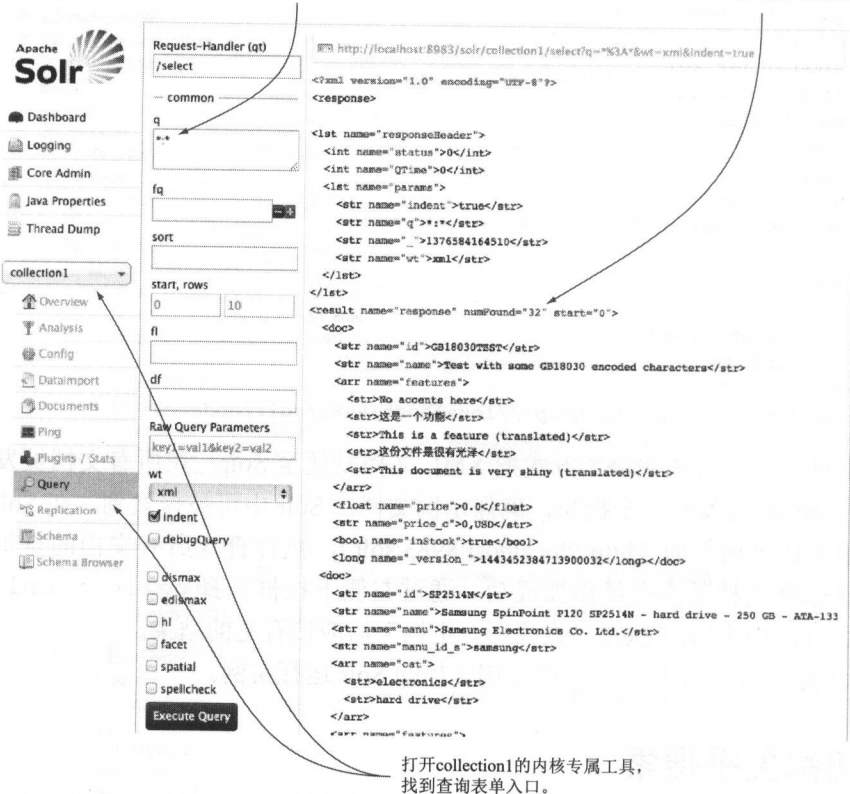


图 2.5 Solr 管理控制台的查询表单截屏。这里可以测试示例文档在执行查找所有文件的查询中是否被正确索引

正如第 1 章 (1.2.3 节) 讨论过的, 所有与 Solr 核心服务有关的交互, 如查询处理, 都是通过 HTTP 请求执行的。填写查询表单之后, 创建一个 HTTP GET 请求并发送给 Solr。表 2.1 中的查询表单字段通过 HTTP GET 请求传递给 Solr 服务器相应的参数。代码清单 2.1 实现的是, 当图 2.6 的查询执行时将 HTTP GET 请求发送给 Solr。请注意, HTTP GET 请求的参数间不包括换行, 这里的换行只是为了更容易理解。

The screenshot shows the Solr Query Form with the following fields and annotations:

- Request-Handler (qt):** /select. Annotation: 第4章将介绍请求处理器。
- common:** (checkbox)
- q:** iPod. Annotation: 主查询字段, 搜索与iPod关键词相关的文档。
- fq:** manu:Belkin. Annotation: 过滤查询 限制搜索结果 集文档的manu字段 取值为Belkin。
- sort:** price asc. Annotation: 搜索结果按price字段 从低到高排序。
- start, rows:** 0, 10. Annotation: 首页从0开始, 每页返回10个搜索结果。
- fl:** name,price,features,score. Annotation: 为搜索结果中每个文档 指定需要返回的字段。
- df:** text.
- Raw Query Parameters:** key1=val1&key2=val2.
- wt:** xml. Annotation: 默认搜索字段。
- indent:** ☒. Annotation: 高级功能的展开选项, 例如, 分面与搜索结果高亮。
- debugQuery:** ☐
- dismax:** ☐
- edismax:** ☐
- hl:** ☐
- facet:** ☐
- spellcheck:** ☐
- wt:** xml. Annotation: 响应输出类型, 例如, XML、CSV或JSON。

图 2.6 Solr 查询表单的注释截屏, 展示了 Solr 查询处理中的主要功能, 例如, 过滤器、搜索结果格式指定、排序、分页与搜索组件

表 2.1 图 2.6 中的查询参数概览表

表单字段	取值	说明
q	iPod	主查询参数; 根据该参数中的词项与文档的相似度, 对文档评分
fq	manu:Belkin	过滤查询; 通过过滤器筛选结果集文档, 但不影响评分。在这个示例中, 我们将制造商字段 manu 限定在 Belkin, 对结果进行筛选
sort	price asc	指定排序字段与排序方式。在这个示例中, 我们希望按照价格字段, 从低到高升序排列文档, 价格最低的排在最前面
start	0	指定搜索结果的起始页; 由于这是第一次请求, 我们希望第一页采用基于 0 的索引方式, 根据页面大小递增到下一页
rows	10	页面大小; 限制每一页显示的结果数量。在这个示例中设置为 10

续表

表单字段	取值	说明
fl	name,price,features,score	在结果集中每个文档返回的字段列表。score 字段是内置字段，用于保存每个文档的查询相关度得分。必须显式地请求 score 字段，它才会返回，如这个示例的做法
df	text	未指定搜索字段的任意查询词项的默认搜索字段；text 是示例服务器的全包含 (catch-all) 字段
wt	xml	响应输出类型；控制响应输出的格式

代码清单 2.1 查询表单的 HTTP GET 请求分解

主要查询组件寻找包含“iPod”的文档。

http://localhost:8983/solr/collection1/select?

- 调用 collection1 内核的 select 请求处理器。
- 过滤出 manu 字段等于 Belkin 的文档。
- 搜索结果中返回 name、price、features 与 score 字段。
- 搜索结果按照价格升序 (从低到高) 排列。
- 默认搜索字段是 text 字段。
- 以 XML 格式返回搜索结果。
- 从第一条搜索结果开始返回，一共返回 10 条搜索结果。

q=iPod&  
fq=manu:Belkin&  
sort=price asc&  
fl=name,price,features,score&  
df=text&  
wt=xml&  
start=0&rows=10

### 需要更多的查询示例吗

第 7 章会深入介绍查询。如果想直接学习更多的实际查询操作，推荐查看 Solr 提供的帮助。在浏览器中打开 `$SOLR_INSTALL/docs/tutorial.html`，其中包含对 2.1.4 节加载的示例文档的更多查询示例。<sup>2</sup>

查询表单并不是为最终用户设计的，而是 Solr 为开发者和管理员提供的一个查询提交途径，让他们无须手动构造 HTTP 请求或专门开发一个客户端程序，就能向 Solr 发送查询请求。但有一点必须澄清，使用 Solr 的应用程序，你需要为其开发用户搜索界面。正如在 2.2.5 节所提到的，Solr 提供了一个名为 Solritas 的可定制的搜索界面样例，帮助你构建优秀的搜索应用。

## 2.2.2 Solr 的搜索返回机制

前面学习了 Solr 的查询请求提交，现在了解一下 Solr 的搜索返回机制。本小节的核心是 Solr 返回与查询匹配的文档集，还包括为达到优质搜索体验 Solr 客户端需要处理的其他信息。搜索体验优化的操作由 Solr 客户端负责。Solr 返回原始的搜索

<sup>2</sup> 译者注 Solr 6 系列版本的教程帮助路径为 `$SOLR_INSTALL/docs/quickstart.html`。



结果数据，并提供核心的搜索功能，要为用户提供优质的搜索体验，则需要根据这些结果和功能进行自主设计与开发。

图 2.7 展示了 2.2.1 节中示例查询所返回的搜索结果。搜索结果是 XML 格式，按照价格从低到高排序。每一个文档均包含 iPod 关键词。因为只有两个搜索结果，所以没有分页。

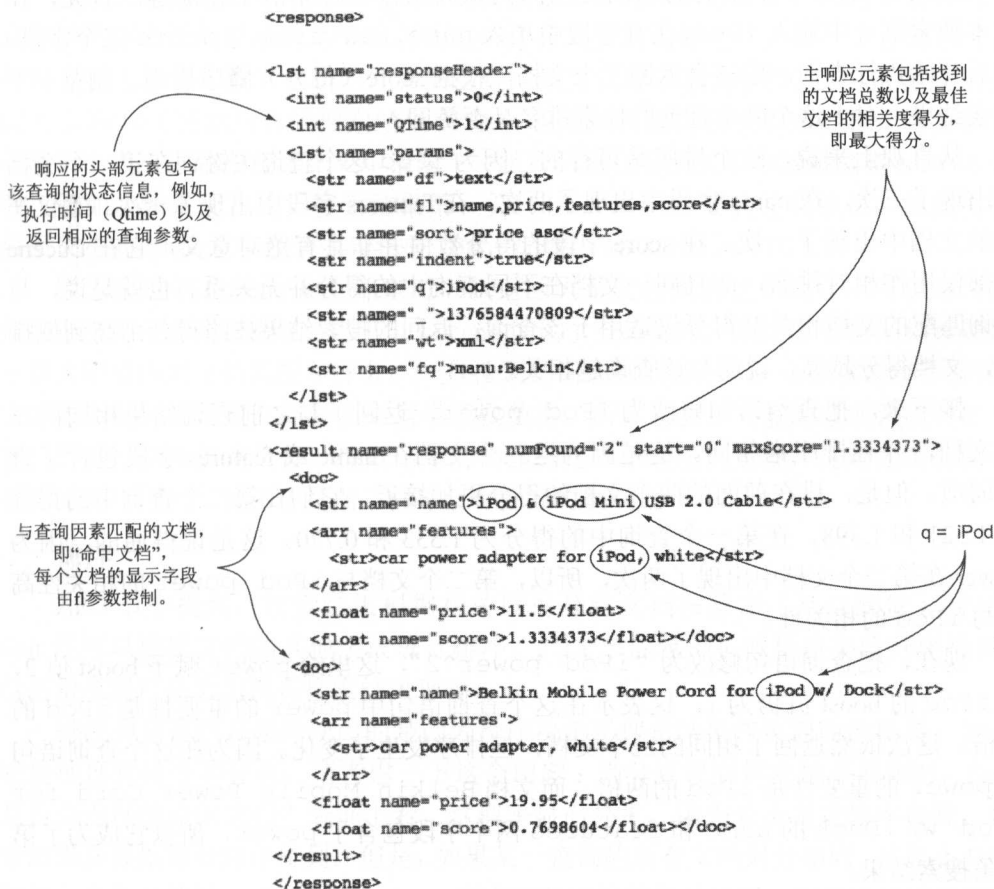


图 2.7 代码清单 2.1 中 Solr 根据请求示例返回的 XML 格式的搜索结果

至此，我们只介绍了 Solr 使用 XML 格式返回搜索结果，其实 Solr 也支持其他格式，例如 CSV（Comma-Separated Values，逗号分隔值）、JSON（JavaScript Object Notation），以及针对主流编程语言的特定格式。例如，Solr 可以返回 Python 专用格式，它允许使用 eval 函数将响应结果正确地解析为一个 Python 对象树。



## 2.2.3 排名检索

第1章提到过，Solr 的查询处理与关系型数据库或其他 NoSQL 数据存储最大的不同就在于排名检索：根据文档与查询的相关性进行排序，最相关的文档将处于列表最前端。

我们以 2.1.4 节中索引的示例文档为例来介绍排名检索的工作原理。首先，在文本搜索框 `q` 中输入 `iPod`，在 `fl` 字段中填入 `name`、`features` 与 `score` 三个字段，然后开始执行查询，应该会返回三个文档，根据 `score`（得分）降序排列。浏览一下搜索结果，对于这个简单查询的检索排名是否赞同？

从直观上来说，这个排序是可行的。因为 `iPod` 这个查询关键词在第一个文档中出现了三次，在 `name` 字段中出现了两次，在 `features` 字段中出现了一次。它仅在其他文档中出现了一次。在 `score` 字段的得分数值并非具有绝对意义，它在 Lucene 内部仅用作相对排名，而且同一文档在不同查询中的得分并无关系。也就是说，与查询匹配的文档相关度得分仅适用于该查询。返回的搜索结果按照得分由高到低排序，文档得分越高，说明与该查询越相关。

接下来，把查询语句修改为 `iPod power`，返回了与之前查询结果相同的三个文档，并且排序也相同。这是因为这三个文档在 `name` 或 `features` 字段包含了查询词项。但是，排在前面的两个文档的得分更加接近，它们在第二个查询中的得分为 1.521 和 1.398，在第一个查询中的得分为 1.333 和 0.770。这是说得通的，因为 `power` 在第二个文档中出现了两次，所以，第二个文档与 `iPod power` 的相关性高于与 `iPod` 的相关性。

现在，把查询语句修改为 `"iPod power^2"`，这里给 `power` 赋予 `boost` 值 2，而 `iPod` 的 `boost` 值仍为 1。这表示在这个查询语句中 `power` 的重要性是 `iPod` 的两倍。这次依然返回了相同的三个文档，但排序发生了变化。因为在这个查询语句中 `power` 的重要性是 `iPod` 的两倍，而文档 `Belkin Mobile Power Cord for iPod w/ Dock` 的 `name` 和 `features` 两个字段包含了 `power`，所以它成为了第一条搜索结果。

现在你应该了解排名检索到底是怎么回事了。第 3、7、16 章会介绍更多有关排名检索与权重调整（`boosting`）的知识。接下来介绍查询处理的其他功能，我们从如何使用分页和排序返回三个以上的文档开始。

## 2.2.4 分页和排序

书中的 Solr 示例索引仅包括 32 个文档。实际上，在生产环境中一个 Solr 实例通常会包含百万数量级的文档。可以把 Solr 实例想象成一个大电子超市，一

个有关 iPod 的查询语句很有可能会匹配上千个产品和配件。为了确保迅速返回搜索结果，尤其是在带宽有限的移动设备上，我们并不希望看到一次返回上千个结果，即使它们都位列前端、十分相关。

## 分页

解决上述问题的方案是使用分页返回搜索结果的小分子集，同时使用导航工具来请求更多页面。分页是 Solr 查询处理中最重要的功能之一，每一个查询包含了控制页面大小（行数）和起始位置的参数。如果查询请求未做指定，Solr 默认的页面大小为 10（页面包含的搜索结果个数），也可以在查询请求中通过指定行参数来调整页面大小。要在搜索结果中请求下一页，只需在页面大小中增量调整 start 参数。例如，如果现在处于搜索结果第一页（start=0），要去往其他页，就需要把页面大小的 start 参数增加为相应的值，如增为 10（start=10）。

因为底层的 Lucene 索引并未对一次返回大量文档做出优化设计，所以尽可能使用小页面是非常重要的。相反，Lucene 优化了查询处理，底层的数据结构设计用于最大限度满足文档匹配和评分。一旦搜索结果被确认后，Solr 必须重建每个文档。大多数情况下，重建文档是从磁盘中读取数据，所以它会尽可能高效地使用智能缓存。然而，相比查询执行，搜索结果的构建仍是一个缓慢的过程，这对大页面而言更是如此。因此，使用小页面，Solr 会有更好的表现。

## 排序

2.2.3 节中提到，搜索结果根据相关度得分将文档按降序（从高到低）排列。Solr 还可以根据文档的其他字段进行排序。2.2.1 节的示例按照价格字段升序排列，价格最低的产品会出现在搜索结果顶端。

排序和分页结合起来使用的是因为排序决定了搜索结果在页面中的位置。为了更好地理解排序和分页，请考虑一下这个问题：当分页没有指定排序原则时，Solr 是否会返回确定的文档？表面上看，这似乎显而易见，因为若不指定排序参数，搜索结果会根据得分降序排列。但是，如果某一查询的所有文档得分相同，该怎么办？例如，查询条件是 inStock:true，所有匹配文档的得分相同，你可以通过查询表单来验证这一情况。

事实证明，即使文档的得分相同，Solr 仍然能以一个确定的顺序返回所有文档。这是因为 Solr 找到与查询相关的所有文档后，会对整个文档集合进行排序和分页。Solr 分别记录了与查询匹配的所有文档的排序和分页。顺便提一下，如果所有文档的得分相同，那么它们会以索引的次序返回，该次序基于 Lucene 的内部文档 ID。这个文档 ID 大致等于被索引文档的次序。但是，由于索引变化时 ID 值会随之变化，所以不应依赖此 ID 进行排序。

## 2.2.5 扩展的搜索功能

查询表单包含复选框列表,可以在查询处理中实现高级功能。如图 2.6 所示 (2.2 节), 查询表单中的各种复选框字段用来激活以下的搜索功能:

- `dismax`——析取最大查询解析器 (参见第 7 章)。
- `edismax`——扩展的析取最大解析器 (参见第 7 章)。
- `hl`——搜索结果高亮 (参见第 9 章)。
- `facet`——分面 (参见第 8 章)。
- `spatial`——地理空间搜索,例如,基于位置距离排序 (参见第 15 章)。
- `spellcheck`——查询词项拼写检查 (参见第 10 章)。

如果随意点击这些复选框,你会发现并不知道接下来该怎么做。在查询表单中使用这些搜索组件需要掌握更多的知识,本章旨在快速入门,所以难以面面俱到,后续章节会详细介绍它们。

现在,我们来看看 Solr<sup>3</sup> 界面的搜索功能。请访问 <http://localhost:8983/solr/collection1/browse> 查看本地 Solr 实例的搜索界面,如图 2.8 所示。<sup>3</sup>

在图 2.8 的顶端, Solr 提供了三个可选样例: **Simple** (简单)、**Spatial** (空间) 和 **Group by** (分组)。这里简要介绍 **Simple** 样例的主要方面,其他两个样例请自行探索。

从图 2.8 中找出搜索相关的多个组件。示例中最为有趣的搜索组件是分面,显示在页面左侧,第一组是字段分面 (Field Facets)。分面组件根据字段取值,将搜索结果划分为有意义的子集,帮助用户进行查询限定和发现新的信息。例如,当搜索 `video` (视频) 时, Solr 返回了三个示例文档, `cat` 字段包含搜索结果的三个子集: `electronics` (3)、`graphics card` (2) 和 `music` (1)。单击 `music` 链接,原有的三个文档被过滤为一个文档。此处的搜索理念是,除了搜索结果,用户的搜索条件通过不同的过滤器进行限定,从而归类搜索结果。第 8 章将详细介绍分面。

---

<sup>3</sup> 译者注 Solr 6 系列版本请访问 <http://localhost:8983/solr/techproducts/browse>。

通过每个示例探索Solr的不同功能。

“More Like This”搜索组件的作用是寻找与搜索结果中该文档相类似的其他文档。

空间搜索组件根据地理位置的远近对文档进行排序。

分页支持。

分面搜索组件将搜索结果的字段值归类为有用的子集。

高亮搜索组件对搜索结果中的查询项进行加粗强调。

The screenshot shows the Apache Solr search interface. At the top, there's a search bar with 'video' entered. Below it, there are tabs for 'Simple', 'Spatial', and 'Group By'. A 'Boost by Price' checkbox is also visible. The search results are displayed in a list format. Each result includes a title, ID, price, features, and stock status. For example, the first result is 'Apple 60 GB iPod with Video Playback Black' with ID MA147LL/A and a price of 309.00,USD. The second result is 'ATI Radeon X1900 XTX 512 MB PCIE Video Card' with ID 100-435805 and a price of 649.99,USD. The third result is 'ASUS Extreme N7800GT/2DHTV (256 MB)' with ID EN7800GT/2DHTV/256M and a price of 479.95,USD. On the left side, there are three facet sections: 'Field Facets' showing categories like 'cat' (electronics, graphics card, music) and 'manu\_exact' (ASUS Computer Inc., ATI Technologies, Apple Computer Inc.); 'Query Facets' showing 'load' and 'GB'; and 'Range Facets' showing 'price' ranges and 'popularity' ranges. Annotations with arrows point to various parts of the interface: one points to the search bar, another to the 'More Like This' link, a third to the spatial search map, a fourth to the pagination, a fifth to the facet sections, a sixth to the 'More Like This' link, and a seventh to the highlighted search terms in the results.

图 2.8 Solr 简单示例，展示了各种搜索组件的使用方法，例如，分面、类似结果、搜索结果高亮和空间搜索，这些为用户提供了丰富的搜索体验

接下来我们看看拼写检查这个搜索组件，它在图 2.8 中并不是那么明显。为了了解拼写检查的工作机制，我们在搜索框中输入 `vydeoh`，取代原先的 `video`。如图 2.9 所示，没有找到任何搜索结果，但 Solr 返回了一个链接，询问用户是否想要搜索 `video`？如果是这样，用户可以通过单击链接重新执行搜索。

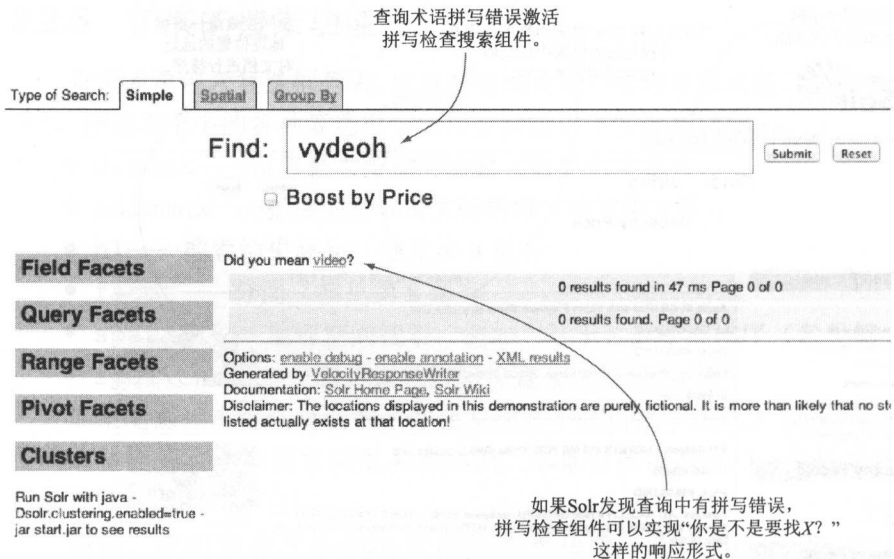


图 2.9 拼写检查组件示例。在搜索界面上提示用户使用查询术语的正确拼写,从而重新执行搜索。在这个示例中, Solr 发现与 vydeoh 最为接近的匹配是 video

Solriatas 的三个样例封装了许多强大的功能,建议你花时间了解每项功能。下面来了解一下管理控制台的其他部分。

## 2.3 Solr管理控制台一览

此时,你应该对查询表单有所了解。接下来,我们快速浏览一下管理控制台的其他部分,如图 2.10 所示。

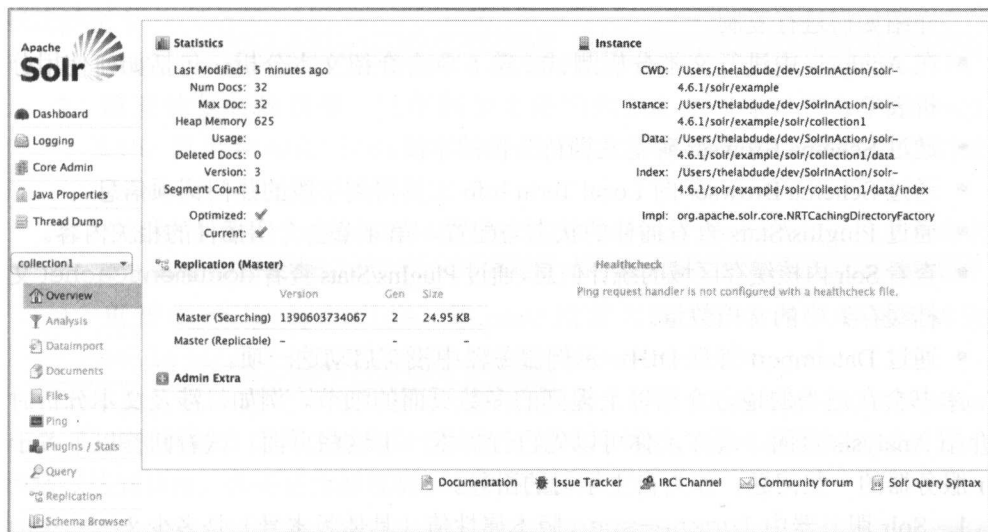


图 2.10 Solr 管理控制台，通过左侧工具栏查看每个页面

与其花时间阅读管理面板的文字，不如自己去尝试点击这些页面。因此，将访问管理控制台所有的链接并熟悉每一页的内容作为练习留给大家。以下是你需要了解的管理控制台的一些特点：

- 了解如何通过仪表盘配置 Solr 实例。
- 从日志中查看最近的日志信息。
- 在日志级别上暂时变更日志的详细设置。
- 在核心管理器中添加并管理多个内核。
- 查看 Java 的系统属性。
- 通过线程转储在 Java 虚拟机 JVM 中获取所有的活动线程。

除了这些主要页面之外，还有一些关于服务器中每个内核的专属页面。之前的示例服务器只有一个名为 collection1 的内核。内核专属页面允许执行以下操作：

- 查看内核的具体属性，例如，主要核心页面（如 collection1）的 Lucene 片段数量。
- 向内核发送一个快速请求，使用 Ping 确认系统正常运行与响应。
- 在特定内核的索引上使用 Query 执行查询语句。
- 在 Schema 中查看该内核当前活跃的 schema.xml。第 5 章和第 6 章会详细介绍 schema.xml。
- 从内核的 Config 中查看当前活跃的 solrconfig.xml。第 4 章会详细介绍 solrconfig.xml。
- 在 Replication 中查看如何将索引复制到其他服务器上。第 12 章和第 13 章会

介绍如何进行复制。

- 在 Analysis 中进行文本分析测试。第 6 章会介绍文本分析, 包括如何使用分析表单。
- 通过 Schema Browser 确定文档待分析的字段。
- 通过 Schema Browser 的 Local Term Info 工具得到字段的热门词项信息。
- 通过 PlugIns/Stats 查看插件的状态与配置。第 4 章会介绍插件的相关内容。
- 查看 Solr 内核缓存区域的统计信息, 通过 PlugIns/Stats 查看 documentCache(文档缓存) 中的文档数量。
- 通过 Dataimport 管理 DIH。示例服务器中没有启动这一项。

本书会在适当的地方介绍以上提到的多数页面的细节。例如, 涉及文本分析时会介绍 Analysis 页面。现在, 你可以先自行探索一下这些页面。试着回答以下关于 Solr 服务器的一些问题, 作为自主学习的引导。

1. Solr 服务器中 lucene-spec 版本属性值 (具体版本号) 是多少?
2. org.apache.solr.core.SolrConfig 类的日志级别是多少?
3. collection1 内核的 maxDoc (最大文档) 属性值是多少?
4. Java.vm.vendor 的 Java 系统属性值是多少?
5. collection1 内核的 segment count (片段数) 是多少?
6. Ping 通服务器的响应时间有多长?
7. manu 字段的热门词项有哪些? (提示: 在 Schema Browser 中选择 manu 字段, 点击 Load Term Info 按钮。)
8. 当前 DocumentCache 的容量有多大? (提示: 考虑 Stats。)
9. name 字段中对 Belkin Mobile Power Cord for iPod w/Dock 进行文本分析, 结果如何? (提示: 在 Analysis 页选择 name 字段。)

接下来, 把注意力放到针对特定需求进行 Solr 定制化时需要完成的工作。

## 2.4 根据需求改造搜索示例服务器

现在有一个操作示例服务器的实战机会, 你可能想知道针对特定需求改造示例服务器的最佳路径。在此, 有两个选择可供参考。其一, 在 example/ 目录基础上进行改动, 以满足你的需求。其二, 更好的选择是先复制一份 example/ 目录, 在 example/ 副本上进行具体应用的定制修改, 这样出现意外问题还能参考原有的 example/ 目录。

如果选择了后者, 需要根据应用的具体情况, 将 example/ 目录修改为适合的名称, 不要直接使用 example/。例如, 对于第 1 章介绍的房地产搜索应用, 我们将其命名为 realestate/。一旦确定了目录名称, 请按照下列步骤在 Solr 中创建一个 example/



目录的副本。

1. 创建 `example/` 目录的完整副本，例如，`cp -R example realstate`。
2. 清理被复制的目录，以便删除未使用的 Solr 主目录。例如，`example-DIH/` 目录和 `multicore/` 目录。如果需要参考它们，可以把它们放在 `example/` 目录下。
3. 在 Solr 主目录下，根据应用的具体情况，将 `collection1/` 目录重命名为更直观的应用名称。
4. 更新 `core.properties`，将 `name` 设置为第 3 步中内核名称相对应的 `collection1` 的新集合名称，例如，`name=realestate`。

请注意，现阶段没有必要修改 Solr 的配置文件，如 `solrconfig.xml` 或 `schema.xml`。这些文件被专门设计以提供良好的开箱即用体验，你可以根据需求反复进行改造以达到目标，而不必全部推翻和重新配置。

### 清理索引

有时需要开启一个全新的索引。Solr 停止后，通过删除内核的 `data/` 目录下的内容，移除所有文档，例如 `solr/collection1/data/*`。重启 Solr，一个不包含任何文档的全新索引就生成了。

2.1.2 节介绍了如何从新的目录重启 Solr。例如，要重启通过 `example/` 复制的 `realestate` 房地产应用，则执行以下命令：

```
cd $SOLR_INSTALL/realestate
java -jar start.jar
```

你可能想要了解如何设置 JVM 选项、配置备份、监控，以及如何将 Solr 作为一个服务来运行等。这些都是在生产环境部署时需要重点关注的。第 12 章会讨论 Solr 生产环境中的这些问题。

## 2.5 本章小结

总结一下，本章一开始介绍了如何从 Apache 官方网站下载并安装 Solr 4.7 的二进制分发版。实际上，安装过程唯一要做的就是，将下载的压缩包（.zip 或 .tgz）解压缩到合适的目录。接下来，启动 Solr 的示例服务器，通过 `post.jar` 命令添加一些示例文档。

然后介绍了 Solr 的查询表单和 Solr 查询的基本组成。具体来说，介绍了查询构建，包括主要查询参数 `q` 和可选过滤器 `fq`；通过 `fl` 参数控制返回的字段和使用 `sort` 控制搜索结果的排序；还介绍了排名检索的原理，搜索结果根据相关度得分进行排

序，接下来第3章会详细讲解。更多有关查询的内容详见第7章。

借助 Solr<sup>1</sup> 样例搜索界面，我们介绍了 Solr 的搜索组件及其工作原理。具体来说，给出了用户使用分面（动态生成的过滤器）对搜索条件进行限定的例子；还介绍了拼写检查组件，当查询中包含拼写错误时，向用户提示“你是不是要找 X？”的提示信息。

接下来，介绍了 Solr 管理控制台的一些可用工具的小贴士。你会找到 Solr 的许多出色的工具和可用的统计信息，希望你在浏览器中熟悉管理控制台界面之后，能够回答书中提出的有关 Solr 服务器的9个问题。本章还介绍了如何在复制 example/ 目录的基础上定制自己的搜索应用。这是一个上手的好方法，因为掌握了它，再根据特定需求去定制化 Solr 时，你就会有一个参照物。

现在 Solr 实例已经运行起来了，接下来我们要学习 Solr 的重要理论知识。第3章会帮助你更好地理解核心搜索概念，这有助于完成后续的 Solr 学习之旅。

# Solr 基础理论

## 本章要点

- Solr 与传统数据库技术的区别
- Solr 内部索引的基本结构
- Solr 如何使用词项、短语与模糊匹配来实现复杂查询
- Solr 如何计算与查询匹配的最相关文档的得分
- 如何在返回相关的结果与尽可能多的结果之间做出权衡
- 如何对内容进行规范化文档建模
- 如何扩展 Solr 服务器集群，用以处理数十亿的文档和查询

现在，Solr 已经启动和运行。了解搜索引擎的基本原理，为什么选择 Solr 来存储检索内容是现阶段的重点。本章的目标是提供理论基础，帮助你更好地理解并充分使用 Solr。

本章内容会帮助你更好地理解后续章节的高级主题，有助于优化用户搜索体验的质量。如果你在信息检索方面不具备坚实的背景知识，最好不要跳过本章。

虽然本章内容对大多数搜索引擎均适用，但我们主要关注信息检索理论在 Solr 中的实践。读完本章，你应该掌握以下知识点：Solr 索引原理；复杂布尔查询与模糊查询的执行；默认相关度的评分模型；如何让 Solr 在多服务器上处理数十亿文档的同时，仍能保持查询速度的架构特征。

首先，讨论 Solr 搜索背后的核心概念，包括搜索索引是怎样工作的，搜索引擎如何对查询与文档进行匹配，Solr 如何通过强大的查询能力去找寻内容——这些在过去可是个难题。

## 3.1 搜索、匹配与找寻内容

现有的许多不同类型的技术系统，如关系型数据库、键值存储、操作磁盘文件的 map-reduce（映射 - 规约）引擎、图数据库等，都是为了帮助用户解决颇具挑战性的数据存储与检索问题而出现的。而搜索引擎，尤其是 Solr，致力于解决一类特定的问题：搜索大量非结构化文本，并返回最相关的搜索结果。

本节介绍现代搜索引擎的核心功能，解释什么是搜索的文档，概述倒排索引（Solr 快速全文搜索能力的核心）及其如何支持各种复杂词项查询、短语查询和部分匹配查询。

### 3.1.1 何为文档

第 2 章中我们向 Solr 发送了一些文档，然后运行了 Solr 示例搜索服务器，所以此处不是第一次提到文档。需要重点掌握的内容是，何种信息类型能够被 Solr 接收并实现搜索，以及这些信息是如何被结构化的。

Solr 是一个文档存储与检索引擎。提交给 Solr 处理的每一份数据都是一个文档。文档可以是一篇新闻报道、一份简历、社交用户信息，甚至是一整本书。

每个文档包含一个或多个字段，每个字段被赋予具体的字段类型：字符串、标记化文本、布尔值、日期/时间、经纬度等。潜在的字段类型数量是无限的，因为一个字段类型是由若干分析步骤组成的，这些步骤会决定数据如何在字段中被处理，以及如何映射到 Solr 索引中。每个字段在 Solr 的 schema 中（第 5 章会讨论）被定义为特定的字段类型，并告知 Solr 接收到此类内容的处理办法。代码清单 3.1 展示了一个示例文档，其中包含每个字段的取值。

代码清单 3.1 Solr 文档示例

```
<doc>
  <field name="id">company123</field>
  <field name="companycity">Atlanta</field>
  <field name="companystate">Georgia</field>
  <field name="companyname">Code Monkeys R Us, LLC</field>
  <field name="companydescription">we write lots of code</field>
  <field name="lastmodified">2013-06-01T15:26:37Z</field>
</doc>
```

要在 Solr 上执行一个查询，我们可以在文档上搜索一个或多个字段，即使字段未包含在该文档中。Solr 将返回那些包含了与查询匹配的字段内容的文档。

值得注意的是，虽然 Solr 为每个文档提供了一个灵活的 schema，但灵活不代表

无模式 (schema-less)。在 Solr 的 schema.xml 中, 所有的字段类型必须被定义, 所有的字段名称 (包括动态字段命名模式) 必须被指定 (第 5 章会进一步讨论)。这并不意味着, 每个文档必须包含所有字段, 仅当所有可能的字段出现在一个文档中需要处理时, 才会被全部映射到特定字段类型中。当首次接收到包含新字段的文档时, Solr 会自动猜测未知的新字段类型。通过检查字段中的数据类型, 自动将字段增加到 Solr 的 schema 中。如果输入的数据难以理解, Solr 可能会对字段类型识别失败, 因此, 更好的做法是预先定义好字段。

一个文档通过定义 schema, 映射为特定字段类型的字段集合, 文档的每个字段根据其字段类型进行内容分析, 分析的结果保存在索引中, 这样在发起查询时就能检索到相关结果。Solr 查询返回的主要搜索结果是由一个或多个字段组成的文档集。

### 3.1.2 基本搜索问题

首先我们来了解一下搜索引擎解决的基本问题, 这有助于深入理解 Solr 的工作原理。

例如, 有一项任务是为读者开发图书搜索功能, 初始原型可能如图 3.1 所示。

**Book Title:**

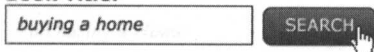


图 3.1 示例搜索界面, 如典型网站的搜索框, 展示了用户如何向搜索应用提交查询

假设读者想购置新房, 输入 “buying a home” 查询相关图书。找到的可能相关的书名如表 3.1 所示。

表 3.1 与 “buying a home” 查询相关的图书

可能相关的图书
The Beginner's Guide to Buying a House
How to Buy Your First House
Purchasing a Home
Becoming a New Home Owner
Buying a New Home
Decorating Your Home

其他一些图书对于有购房打算的客户而言并不相关, 书名列表如表 3.2 所示。

表 3.2 与“buying a home”查询不相关的图书

不相关的图书
A Fun Guide to Cooking
How to Raise a Child
Buying a New Car

传统 SQL 数据库实现该查询的 SQL 语句方式如下，即匹配精确的文本。

```
SELECT * FROM Books
WHERE Name = 'buying a new home';
```

这种方式的问题在于，SQL 查询会对用户输入进行精确匹配，如果找不到任何精确匹配的结果，则在图书目录中显示找不到相关的书名。只有查询能够精确匹配到完整书名，用户才能看到相应结果。

一种更为模糊的处理方式是搜索用户查询中的单个词，同时包含这三个词的书名就会被返回。

```
SELECT * FROM Books
WHERE Name LIKE '%buying%'
      AND Name LIKE '%a%'
      AND Name LIKE '%home%';
```

以上 SQL 查询没有利用现有的数据库索引，在传统数据库中的处理花销较高，但至少得到了一个包含所有查询词汇的匹配结果，如表 3.3 所示。

表 3.3 数据库 LIKE 查询，对每个词进行模糊匹配

匹配的图书	不匹配的图书
<i>Buying a New Home</i>	<i>The Beginner's Guide to Buying a House</i>
	<i>How to Buy Your First House</i>
	<i>Purchasing a Home</i>
	<i>Becoming a New Home Owner</i>
	<i>A Fun Guide to Cooking</i>
	<i>How to Raise a Child</i>
	<i>Buying a New Car</i>
	<i>Decorating Your Home</i>

当然，你可能会觉得在查询中匹配用户输入的所有词这个要求仍然过于严格。让搜索体验变得更加灵活的一种简单方法是，只要一个词出现在书名里即认为匹配成功，SQL 查询语句如下：

```
SELECT * FROM Books
WHERE Name LIKE '%buying%'
      OR Name LIKE '%a%'
      OR Name LIKE '%home%';
```

该查询结果如表 3.4 所示。与前一个查询相比，该查询匹配到更多的书名。这是因为查询仅仅要求匹配到一个关键词即可。另外，由于该查询对每个关键词仅作部分字符串匹配，任何包含字母“a”的书名也被返回了。先前的示例也匹配了字母“a”，但因为要求其他几个关键词同时存在，所以并没有出现返回大量结果的问题。

表 3.4 数据库 LIKE 查询——仅要求匹配至少一个词的结果

匹配的图书	不匹配的图书
<i>A Fun Guide to Cooking</i>	<i>How to Buy Your First House</i>
<i>Decorating Your <b>Home</b></i>	
<i>How to Raise <b>a</b> Child</i>	
<i><b>Buying</b> a New Car</i>	
<i><b>Buying</b> a New <b>Home</b></i>	
<i>The Beginner's Guide to <b>Buying</b> a House</i>	
<i>Purchasing <b>a</b> <b>Home</b></i>	
<i>Becoming <b>a</b> New <b>Home</b> owner</i>	

第一个查询要求所有词都匹配到，这导致一些相关的图书未被找到；第二个查询要求匹配到一个词即可，这导致过多的图书被找到，同时也出现了许多不相关的图书。

这些示例说明了查询执行中存在一些困难：

- 仅执行子字符串匹配，不能区分出词。
- 不能理解语言变体，例如“buying”与“buy”。
- 不能理解同义词，例如“buying”与“purchasing”、“home”与“house”。
- 类似“a”这样不重要的词汇会影响到预期的搜索结果（相关结果被排除在外或不相关结果的出现，取决于这些词是要“全部”还是“任意”匹配）。
- 结果的相关度排序是无意义的。仅匹配到一个查询词的图书比匹配到多个查询词的图书排名更靠前。

当书目数量越来越大且用户查询量不断增长时，这些查询执行起来会变得越来越慢。这是由于查询必须扫描所有的书名，以便找到部分匹配，而不是使用索引来查找词汇。

以 Solr 为代表的搜索引擎在解决此类问题方面可以大展身手。Solr 会对内容和查询进行文本分析，确定文本相似的词，理解并匹配同义词，移除“a”、“the”和“of”



这类不重要的词，每个搜索结果的得分是基于它与查询词的匹配程度来计算的，以确保最佳结果排在前面，用户无须为找寻期望的内容而翻阅大量不相关的页面结果。Solr 之所以能完成以上工作，是因为使用了索引将内容映射至文档的方式。这与传统数据库模型——文档映射至内容的方式不同。倒排索引是搜索引擎运作的核心。

3.1.3 倒排索引

Solr 使用 Lucene 倒排索引来驱动快速搜索功能，并且在查询时提供了许多其他附加功能。虽然本书不会过多讲解 Lucene 内部数据结构，但它对理解倒排索引的大体结构来说非常重要。若想深入了解，推荐阅读 Michael McCandless、Erik Hatcher 和 Otis Gospodnetić 合著的 *Lucene in Action; Sencond Edition* (Manning 出版社于 2010 年出版)。

回顾一下之前的图书搜索示例，感受一下索引中词项如何映射至文档，如表 3.5 所示。

表 3.5 多个文档映射为一个倒排索引。左侧是原始文档，右侧的倒排索引里包含每一个术语，以及它们在原始文档中的位置

原始文档		Lucene 倒排索引			
文档编号	内容字段	词项	文档编号	词项	文档编号
1	A Fun Guide to Cooking	a	1,3,4,5,6,7,8	...	...
2	Decorating Your Home	becoming	8	guide	1,6
3	How to Raise a Child	beginner's	6	home	2,5,7,8
4	Buying a New Car	buy	9	house	6,9
5	Buying a New Home	buying	4,5,6	how	3,9
6	The Beginner's Guide to	car	4	new	4,5,8
	Buying a House	child	3	owner	8
7	Purchasing a Home	cooking	1	purchasing	7
8	Becoming a New Home Owner	decorating	2	raise	3
9	How to Buy Your First House	first	9	the	6
		fun	1	to	1,6,9
		...	...	your	2,9

在传统数据库中，多个文档的表征是一个文档 ID 对应一个或多个内容字段，这些字段包含文档中出现的单词 / 词项。倒排索引将这个模型颠倒过来了，将语料库中的每个单词 / 词项与它们出现的文档对应起来。在表 3.5 中，在插入倒排索引

之前，原始输入的文本会根据空格进行拆分，每个词项被转换成小写，其他部分保持不变。值得注意的是，实际中可能还需要很多其他的文本转换，不仅仅有这里提到的简单处理。在内容分析处理中词项可以被修改、添加或移除，第 6 章会详细介绍。

最后，应注意倒排索引的两个重要细节：

- 倒排索引中的所有词项对应一个或多个文档。
- 倒排索引中的词项根据字典顺序升序排列。

这里是对倒排索引的简要介绍，3.1.6 节会提到索引中存储的其他信息，用以改进 Solr 查询与评分机制。

下一小节会介绍 Lucene 倒排索引结构，它提供了许多强大的查询功能，尽可能提升关键词搜索的速度和灵活性。

### 3.1.4 词项、短语与布尔逻辑

现在我们已经了解了 Lucene 倒排索引的构成，接下来分析查询如何使用索引找到匹配的文档。本节简要介绍倒排索引中如何查找词项与短语，并运用布尔逻辑和模糊查询来扩展查询功能。参考之前的图书搜索示例，这里使用一个简单查询“new house”，如图 3.2 所示。

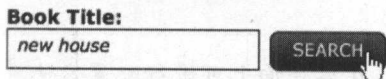


图 3.2 查询解释的细微差别演示

上一小节提到，插入倒排索引时，内容字段的所有文本内容会被分解成单个的词项。现在有一个传入的查询，需要通过一些选项来查询该索引：

- 搜索两个不同的词项，new 和 house，要求两者都被匹配。
- 搜索两个不同的词项，new 和 house，要求匹配其一即可。
- 搜索短语 "new house"，要求精确匹配。

根据用例的实际情况，这几种选项都可能成为有效的方案。受益于 Solr 的强大查询功能和基于 Lucene 的索引构建，布尔逻辑查询很容易实现。

#### 必备词项

先来看第一种选项，将查询分解为多个词项，要求匹配所有词项。使用 Solr 的默认查询解析器编写该查询，有如下两种相似的方法：

- +new +house
- new AND house

这两种方法在逻辑上是相同的。事实上，第二个示例解析后可以约简到第一个

示例形式。符号 + 是一元运算符，要求其后的查询部分必须出现在匹配到的文档中；关键词 AND 是二元操作符，要求其左右两端的查询词必须同时出现。

### 可选词项

与 AND 操作符相对，Solr 还支持 OR 二元运算符，它要求左右两端的查询词至少有一个出现在匹配到的文档中。Solr 默认配置为查询词之间是 OR 运算，无须指定可选参数。以下两个查询是等价的：

- new house
- new OR house

### 排除词项

除了查询词可选或必备之外，还有一种情况是要求词项不出现在匹配到的文档中。以下两个查询是等价的：

- new house -rental
- new house NOT rental

这两条查询的意图是，在匹配 new 或 house 的查询文档中删除包含 rental 的文档。

### Solr 默认运算符

Solr 的默认配置是将词项或短语视为可选的，在单查询上可进行配置，使用 URL 里的 q.op 参数配置多种查询句柄。

```
/select/?q=new house&q.op=OR versus /select?q=new house&q.op=AND
```

请注意，如果把默认运算符 OR 修改为 AND，那么无须指定布尔运算就会要求匹配所有的词项。对于查询 new house，如果默认运算符是 OR，匹配到其中一个词项即可。如果默认运算符是 AND，则要求同时匹配这两个词项。明确指定词项之间的运算符（例如，new AND home 或 new OR home）可以覆盖默认的运算符。

### 短语

Solr 不仅支持单个词项搜索，还支持短语搜索，确保多个词项按特定的顺序出现：

- "new home" OR "new house"
- "3 bedrooms" AND "walk in closet" AND "granite countertops"

### 组合表达式

之前介绍的都是查询表达式，最后一种是使用布尔逻辑对词项、短语和其他查询表达式进行组合构造。Solr 查询语法使用括号组合词项的方式来构造任意复杂的查询表达式，具体如下：

- New AND (house OR (home NOT improvement NOT depot NOT grown))
- (+ (buying purchasing -renting) + (home house residence - (+property -bedroom)))

必备词项、可选词项、排除词项与组合表达式提供了强大且灵活的查询功能集，可以对索引进行各种复杂的查询操作，下一小节会介绍。

### 3.1.5 找到文档集

对词项、短语与布尔查询有了基本认识之后，我们现在来研究 Solr 如何使用 Lucene 内部倒排索引来查找匹配的文档。回顾表 3.5 的书名索引，其中一部分也出现在了表 3.6 中。

表 3.6 书名集合的倒排索引

词项	文档编号	词项	文档编号
a	1,3,4,5,6,7,8	...	...
becoming	8	guide	1,6
beginner' s	6	home	2,5,7,8
buy	9	house	6,9
buying	4,5,6	how	3,9
car	4	new	4,5,8
child	3	owner	8
cooking	1	purchasing	7
decorating	2	raise	3
first	9	the	6
fun	1	to	1,6,9
...	...	your	2,9

如果用户查询 new home，Solr 如何在先前的倒排索引基础上准确地找到与该查询匹配的文档呢？

答案是，查询 new home 包含两个词项（new 和 home 之间有一个默认运算符，还记得吗？），在 Lucene 索引中分别查找两个词项。

词项	文档编号
home	2,5,7,8
new	4,5,8

一旦形成每个词项的匹配文档列表之后，Lucene 就会执行集合操作，得到与该查询匹配的适合的结果集。假设默认的运算符是 OR，那么查询的结果就是两个查询词结果的并集，如图 3.3 所示的文氏图。

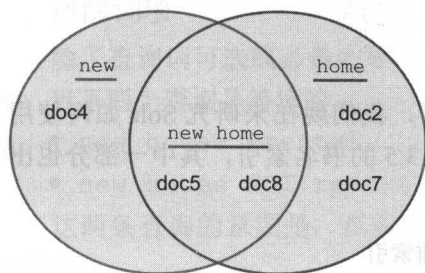


图 3.3 使用 OR 运算符得到查询结果的并集

同样地，如果查询为 new AND home 或默认运算符设为 AND，则查询结果为两个词项查询的结果交集，即返回文档 5 和文档 8，如图 3.4 所示。

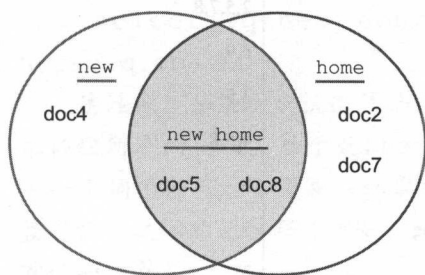


图 3.4 使用 AND 运算符得到查询结果的交集

除了并集与交集查询，排除特定词项也很常见。图 3.5 展示了两个词项各种不同逻辑搭配得到的结果（假设默认 OR 运算符）。

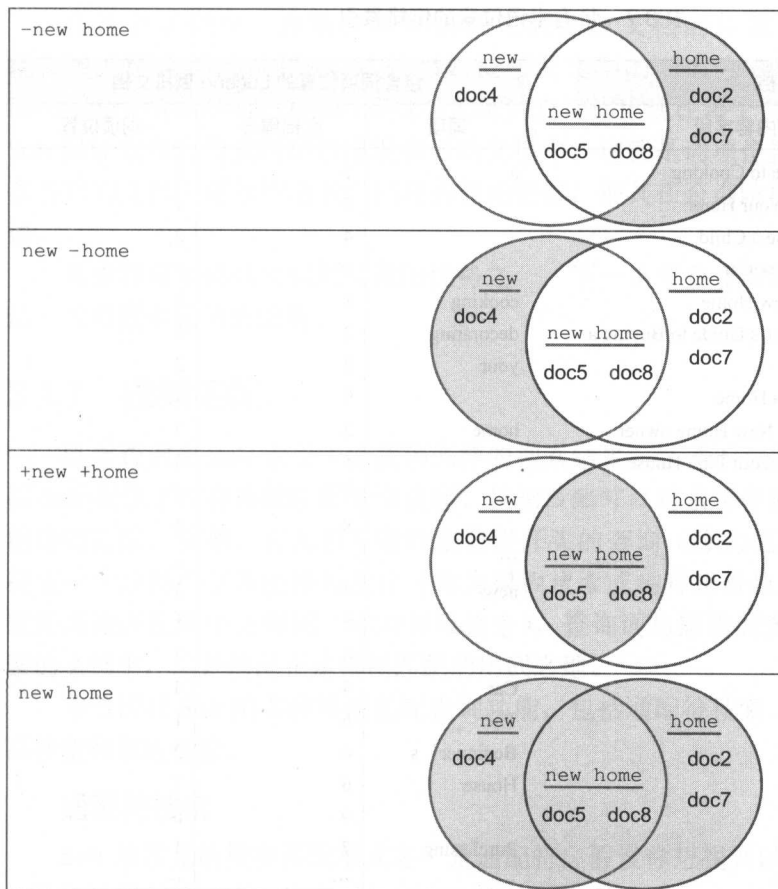


图 3.5 常见布尔查询运算的图形化表示

如上所述，必备词项、可选词项、排除词项和组合表达式的搜索功能为单关键词的查询提供了强有力的支持。下一小节介绍多词项的短语查询功能。

### 3.1.6 短语查询与术语位置

如前所述，在 Lucene 索引上除了可以查询词项之外，还可以查询短语。但是索引只包含单个的词项，那么如何搜索完整的短语呢？

简单的答案是，短语中的每个词项依然在 Lucene 索引中分别检索，就好像提交的查询是两个查询词组合 new home，而不是 "new home" 整个短语。一旦发现重叠的文档集，有一项之前未讨论过的倒排索引特征就会开始发挥作用，这项特征就是词项位置，它会记录词项在文档中的相对位置，这是一个可选项。表 3.7 展示了文档（表的左侧）如何映射到带有词项位置（表的右侧）的倒排索引。

表 3.7 带有术语位置的倒排索引

原始文档		包含词项位置的 Lucene 倒排文档		
文档编号	内容字段	词项	文档编号	词项位置
1	A Fun Guide to Cooking	a	1	1
2	Decorating Your Home		3	4
3	How to Raise a Child		4	2
4	Buying a New Car		...	...
5	Buying a New Home	cooking	1	5
6	The Beginner's Guide to Buying a House	decorating	2	1
7	Purchasing a Home	your	2	2
8	Becoming a New Home owner	home	9	4
9	How to Buy Your First House		2	3
			5	4
			7	3
			8	4
		...	...	...
		new	4	3
			5	3
			8	3
		Car	4	4
		The	6	1
		Beginner' s	6	2
		House	6	7
			9	6
		Purchasing	7	1
		...	...	...

从表 3.7 的倒排索引可以看出, 查询 new AND home 的搜索结果是文档 5 和文档 8。词项位置更进一步指出文档中每个词项出现的位置。表 3.8 显示了一个简略版的倒排索引, 仅显示了 new 和 home 两个词项的交集。

表 3.8 带有词项位置的简略版倒排索引

词项	文档编号	词项位置
home	5	4
	8	4
new	5	3
	8	3



在这个示例中，查询词 new 出现在被匹配文档的位置 3，查询词 home 出现在被匹配文档的位置 4。这是因为被匹配文档的书名为 *Buying a New Home* 和 *Becoming a New Home Owner*。确保被匹配的查询词出现在每个文档中的一个位置上，Solr 保证查询词构成的短语出现在原始文档中。这就是词项位置的强大之处，允许在各自的文档中重新构造索引词项的初始位置，使其在查询阶段可以搜索特定的短语。

搜索特定短语只是词项位置的优势之一。下一小节会介绍词项位置的另一种用法：改进搜索结果的质量。

### 3.1.7 模糊匹配

既定查询在 Solr 索引中能找到怎样的搜索结果，这并不总能提前准确知晓，所以 Solr 提供了几种模糊匹配查询功能。模糊匹配可以对索引中的词项进行并不那么精确的匹配。例如，有人想搜索特定前缀开头的查询（称为通配符搜索），有人想搜索一个或两个字符的拼写变化（称为模糊搜索或编辑距离搜索），有人想根据特定距离来匹配两个查询词（称为邻近搜索）。查询词与短语的多种变化存在于被搜索的文档中，这种情况正是模糊匹配的用武之地。

本节探讨 Solr 的多种模糊匹配查询功能，包括通配符搜索、区间搜索、编辑距离搜索和邻近搜索。

#### 通配符搜索

Solr 最常见的模糊匹配形式之一是通配符。假设你想找到以 offic 开头的文档，一种方法是创建一个查询，列举所有可能的词汇形式。

- 查询：office 或 officer 或 official……

在查询中罗列所需的词汇，这显然对用户来说是不合理的。

由于能匹配到的词汇变化已经存在于 Solr 的索引中，所以我们可以使用星号(\*)通配符实现相同的查询功能，如下所示。

- 查询：offi\* 匹配 office、officer 及 official 等。

除了用在匹配查询词的末尾，通配符还可用于搜索词内部，例如，同时匹配 office 和 offer，则查询如下。

- 查询：off\*r 匹配 offer、officer 及 officiator 等。

星号(\*)通配符匹配查询词中 0 个或多个字符。若只匹配单个字符，可以使用问号(?)来实现。

- 查询：off?r 匹配 offer，但不匹配 officer。

### 首位通配符

虽然 Solr 的通配符功能相当强大,但特定的通配符查询执行可能花销巨大。当通配符搜索执行时,倒排索引中的所有词项与第一个通配符之前的查询词部分进行匹配。接下来,检查每个候选词项是否与查询中的通配符模式相匹配。正因为如此,在通配符之前指定越多的字符,查询执行速度越快。例如,查询 `engineer*` 的执行花销不大,这是因为倒排索引里只能匹配到很少的词项。又如,查询 `e*` 的执行花销会很大,因为它会匹配以字母 `e` 开头的所有词项。

执行首位通配符查询是一项花销甚大的操作。例如,假设你需要匹配以 `ing` 结尾的所有词项(例如 `caring`、`liking` 和 `smiling`),那可能会导致严重的性能问题。

- 查询: `*ing`

如果需要在搜索中使用首位通配符,其实还有一种更快的解决方案,但是需要做一些额外的配置。具体方法是在字段类型的分析链中增加 `ReversedWildcardFilterFactory` 类,有关文本处理配置将在第 6 章中讨论。

`ReversedWildcardFilterFactory` 在 Solr 索引中两次插入被索引的内容,一次是每个词项的文本,另一次是每个词项的反向文本。

- 索引 `caring`      `liking`      `smiling`  
          `#gnirac`    `#gnikil`    `#gnilims`

带有 `*ing` 首位通配符的查询提交后, Solr 会搜索反向版本,将首位通配符搜索转化成标准通配符搜索,这会在反向内容中导致性能问题。

请注意,在 Solr 索引中开启对所有词项的双索引,会增加索引的大小且拖慢整体搜索效率。所以除非有必要,否则不建议开启这个功能。

使用通配符搜索还需要注意一点,即通配符只适用于单个查询词,不适用短语搜索,举例如下。

- 适用: `softwar* eng?neering`
- 不适用: `"softwar* eng?neering"`

若要在短语内执行通配符搜索,就需要在索引中将整个短语存为单个词项,第 6 章最后会介绍处理方法。

### 区间搜索

Solr 还提供了在已知区间值中进行搜索的功能,适用于在一个区间内搜索特定文档子集。例如,若要匹配 2012 年 2 月 2 日到 2012 年 8 月 2 日期间创建的文档,则可以执行以下搜索:

- 查询: `created:[2012-02-01T00:00.0Z TO 2012-08-02T00:00.0Z]`

该区间查询格式也适用于其他字段类型。

- 查询: `yearsOld:[18 TO 21]` 匹配 18、19、20、21。
- 查询: `title:[boat TO boulder]` 匹配 boat、boil、book、boulde 等。
- 查询: `price:[12.99 TO 14.99]` 匹配 12.99、13.000009、14.99 等。

区间查询使用方括号, 这是“包含边界值”的语法。另外, Solr 还支持使用大括号实现不包含区间边界值的搜索, 举例如下。

- 查询: `yearsOld:{18 TO 21}` 匹配 19、20, 但不匹配 18 或 21。

Solr 还支持方括号和大括号的组合搭配, 以形成更加灵活的区间搜索, 尽管语法看起来很奇怪。

- 查询: `yearsOld:[18 TO 21}` 匹配 18、19、20, 但不匹配 21。

虽然区间搜索的执行比单个词项搜索要慢, 但它为 Solr 索引上动态匹配一组取值的特定区间提供了更大的灵活性。请注意, 区间查询的词项排序就是 Solr 索引中被找到的顺序, 即字典顺序。如果创建的文本字段包含整数, 这些整数将按照 1、11、111、12、120、13 这样的顺序排列。Solr 的数值字段会以一种特殊的方式进行索引, 随后章节会提到。这里要理解一点, Solr 索引的排序取决于当时索引创建时的字段数据的处理方式。第 5 章和第 6 章会深入探讨内容分析方式。

### 模糊 / 编辑距离搜索

对于多数搜索应用而言, 精确匹配用户输入的文本, 灵活处理拼写错误, 甚至对正确拼写做出细微修正都是非常重要的。Solr 基于 Damerau-Levenshtein 距离的编辑距离度量方法提供了字符变体的处理手段, 可以有效解决 80% 以上的人为拼写错误。<sup>1</sup>

Solr 使用 ~ 符号表示模糊编辑距离搜索, 如下所示。

- 查询: `administrator~` 匹配 administrator、administrater、administratior 等。

这个查询匹配到原始词项 (administrator) 及与之相距两个编辑距离的其他词项。编辑距离被定义为字符的一次插入、删除、替换或位置互换 (transposition)。Adminstrator (在第 6 个位置上缺少 “i”) 相当于删除 administrator 中的一个字符, 因而它与 administrator 之间存在一个编辑距离。同样地, sadministrator 与 administrator 之间存在一个编辑距离, 因为在 administrator 前面插入了一个 “s”; administratro 与 administrator 之间存在一个编辑距离, 因为对 administrator 的最后两个字符做了调换, “or” 变为了 “ro”。

---

1 Fred J. Damerau, “A Technique for Computer Detection and Correction of Spelling Errors,” *Communications of the ACM*, 7(3):171-176 (1964).

另外，可以修改编辑距离的尺度，让搜索可以匹配任意编辑距离的词条。

- 查询：`administrator~1` 匹配 1 个以内的编辑距离。
- 查询：`administrator~2` 匹配 2 个以内的编辑距离（如果不指定编辑距离，这是默认值）。
- 查询：`administrator~N` 匹配  $N$  个以内的编辑距离。

请注意，两个以上的编辑距离会使得搜索速度大幅下降，也可能匹配出意外的词条。一到两个编辑距离的术语搜索使用有效的 Levenshtein 自动机方法执行，但超过两个编辑距离的查询就会退回到更慢的编辑距离实现方法。

### 邻近搜索

在上一小节里，我们看到编辑距离查找到的词条很接近原始词条，但并不是一模一样的。编辑距离原则上适用于词条字符的替换及短语内词条的变形。

例如，在公司员工档案中搜索高管，有一种实现方法是列举所有包括“executive”的头衔，如下。

- 查询：`"chief executive officer" OR "chief financial officer" OR "chief marketing officer" OR "chief technology officer" OR.....`

当然，这里假设你知道所有可能的头衔，这种方法在其他环境中并不适合。另一种方法是独立搜索每一个词条。

- 查询：`chief AND officer`

这会匹配所有可能的情况，但同时还会匹配到一些无意义的文档——在文档的别处包含了这些词语，例如：`One chief concern arising from the incident was the safety of the police officer on duty.` 这个文档与我们的搜索需求基本没有关系，但在之前的查询中它会被返回。

幸好，Solr 对此提供了解决办法：邻近搜索。针对前面出现的问题，一种好办法是让 Solr 返回 `chief` 与 `officer` 紧临的文档，以下是查询语句。

- 查询：`"chief officer"~1`
  - 含义：`chief` 与 `officer` 之间最多可以相隔 1 个词。
  - 示例：`"chief executive officer"`、`"chief financial officer"`。
- 查询：`"chief officer"~2`
  - 含义：`chief` 与 `officer` 之间最多可以相隔 2 个词。
  - 示例：`"chief business development officer"`、`"officer chief"`。
- 查询：`"chief officer"~N`
  - 含义：`chief` 与 `officer` 之间最多可以相隔  $N$  个词。

之前的邻近查询可视为传统短语搜索的“马虎”版本。事实上，“chief development officer”可以被很容易地重写为“chief development officer”~0。由于编辑距离为0等价于精确短语搜索，所以二者的搜索结果相同。这两种机制均运用到了存储在 Solr 索引中的词项位置（3.1.6 节提到过）。值得注意的是，Solr 邻近查询并未真正使用编辑距离，因为它要求全部特定词项出现，而真正的编辑距离允许替换和删除（对单一词项进行模糊查询）。

编辑距离的一般原则仍然适用于词项邻近查询中的词项插入与换位。沿着这条思路，你可能也注意到，为了能够匹配到“office chief”，需要指定短语间距值（slop）为2，即“chief office”~2。这是因为，第一个编辑操作把 chief 和 office 两个词移动到相同的位置，第二个编辑操作把 chief 向 office 左侧移动一个位置。这里再次强调了一个事实，邻近搜索并不适用真实的编辑距离，换位可能只算作一次编辑操作。那么会有这样的问题出现：“如果要在邻近查询中准确无误地指定短语，那得需要搜集多少个位置信息，把它们添加到文档的文本呢？”

### 3.1.8 快速小结

至此，你应该已经掌握了 Solr 如何在倒排索引中存储信息和查找匹配的文档，包括检索词项，运用布尔逻辑创建各种复杂查询，基于词项对应结果集的集合运算得到搜索结果。另外，本节还讨论了 Solr 如何存储词项位置并通过词项位置查找精确的短语，以及使用邻近查询与位置计算实现模糊短语匹配。对于单词项的模糊搜索，我们使用通配符和编辑距离搜索找到拼写错误和相似的单词。第7章会详细介绍 Solr 的搜索功能，这些核心操作构成了绝大多数 Solr 查询的基础。本节内容为下一节 Solr 关键词的相关度评分模型奠定了基础。

## 3.2 相关度

寻找匹配的文档是构建优质搜索体验的关键步骤，但这仅仅是第一步。大多数用户不愿意通过逐页翻阅搜索结果来找到想要的文档。根据一般经验估计，仅有10%的用户在网页搜索中有意愿继续翻阅第一页以后的搜索结果，仅有1%的用户会翻看到第三页结果。

Solr 出色地实现了搜索结果排序中最佳匹配的文档位于搜索结果列表顶端，这是它的开箱即用功能之一。它会计算每个文档的相关度得分，并从最高分到最低分对搜索结果进行排序。本节介绍相关度得分的计算方法及影响得分的因素。我们将深入探讨 Solr 默认的相关度计算原理和相关度得分的计算公式，并提供直观的示例，帮助你在读完本节之后打下扎实的基础。对大多数人而言，这部分是 Solr 中最难理

解的内容。我们从 Similarity 类开始讨论,它负责查询相关度得分计算的主要方面。

### 3.2.1 默认相似度

Solr 的相关度得分是基于 Similarity 类的。在 Solr 的 schema.xml 中,这个类被定义为一个预置字段(第 5 章将具体讨论)。Similarity 是一个 Java 类,它根据给定查询定义了搜索结果相关度得分的计算方法。你可以选择多个 Similarity 类,甚至编写自己的 Similarity 类,但在此之前,你需要理解 Solr 默认的 Similarity 实现及其理论基础,这是非常重要的。

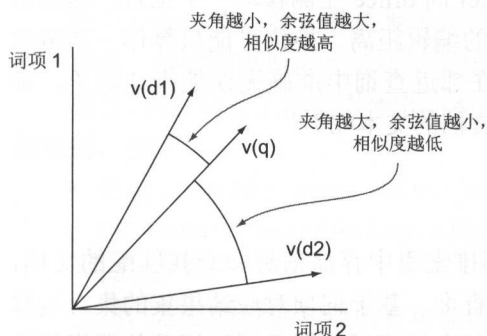


图 3.6 词项向量的余弦相似度。查询的词项向量  $v(q)$  和文档 1 的词项向量  $v(d1)$  更接近,与文档 2 的词项向量  $v(d2)$  相对较远。这是通过计算每个文档的词项向量与查询的词项向量之间的余弦夹角得出的。夹角越小,该文档就被视为与查询的词项向量越相似

默认情况下, Solr 使用 Lucene 相应的 DefaultSimilarity 类。这个类使用两段式检索模型来计算相似度。首先,使用布尔模型(3.1 节介绍过)过滤出不符合用户查询的所有文档。然后,使用向量空间模型通过计算和绘制将查询和文档转换为向量,在此基础上计算相似度得分。每个文档的相似度得分基于查询向量与文档向量的夹角余弦值,3.6 节会介绍。

在向量空间评分模型中,每个文档的词项向量经过计算后会与给定查询对应的词项向量进行比较。两者的相似度通过夹角的余弦值来表示,余弦值为 1,表示完美匹配;余弦值为 0,表示无相似性。更直观地说,两个向量越接近(如图 3.6 所示),则表示它们的相似性越高,也就是说,两向量之间的夹角越小,即余弦值越大,则表示匹配度越高。

当然,如何针对查询和文档的重要特征构造合理的向量来表征它们,继而进行比较,这是整个过程中最具挑战性的任务。我们来看一下 DefaultSimilarity 类的相关度计算公式,依次对公式中每一部分的作用进行直观解释。



给定查询 (q) 和文档 (d)，查询对应的文档相似度的得分计算如图 3.7 所示。

Score (q,d) =

$$\sum_{t \text{ in } q} \left( \text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t,d) \right) \cdot \text{coord}(q,d) \cdot \text{queryNorm}(q)$$

其中:

t= 词项; d= 文档; q = 查询; f = 字段

$\text{tf}(t \text{ in } d) = \text{numTermOccurrencesInDocument}^{1/2}$

$\text{idf}(t) = 1 + \log(\text{numDocs} / (\text{docFreq} + 1))$

$\text{coord}(q,d) = \text{numTermsInDocumentFromQuery} / \text{numTermsInQuery}$

$\text{queryNorm}(q) = 1 / (\text{sumOfSquaredWeights})^{1/2}$

$\text{sumOfSquaredWeights} = q.\text{getBoost}()^2 \cdot \sum_{t \text{ in } q} (\text{idf}(t) \cdot t.\text{getBoost}())^2$

$\text{norm}(t,d) = d.\text{getBoost}() \cdot \text{lengthNorm}(f) \cdot f.\text{getBoost}()$

图 3.7 DefaultSimilarity 评分算法。接下来各小节会具体介绍公式的每个部分

这个公式第一眼看上去非常复杂。不过将公式分解之后就比较直观了。数学的东西仅供参考，除非你决定要为自己的搜索应用重写 Similarity 类，否则没有必要深究整个公式。

相关度计算的主要概念在图 3.7 的抽象公式中都有所体现，包括词项频次 (term frequency, tf)、反向文档频次 (inverse document frequency, idf)、词项权重 (t.getBoost)、字段规范化 (norm)、协调因子 (coord) 和查询规范化 (queryNorm)。接下来将一一介绍。

### 3.2.2 词项频次

词项频次是指特定词项在待匹配文档中出现的次数，表示了文档与该词项的匹配程度。

假如在新闻报纸的索引上搜索一篇关于美国总统的文章，你更想要得到只只提及了一次总统的文章，还是通篇讨论总统的文章？如果某篇文章中 President 和 United States 各出现一次（有可能彼此脱离了上下文），这能说明什么吗？如果一篇文章中这些短语多次出现，是否就该被视为相关呢？

从表 3.9 明显可以看出，第二篇文章比第一篇文章更相关。在第二篇文章中，President 与 United States 多次出现，这表明它的内容与此查询更接近。



表 3.9 提到 President 和 United States 的文档

文章 1 ( 较不相关 )	文章 2 ( 较为相关 )
Dr. Kohrt is the interim president of Furman University, one of the top liberal arts universities in the southern United States. In 2011, Furman was ranked the 2nd most rigorous college in the country by Newsweek magazine, behind St. John’ s College (NM). Furman also consistently ranks among the most beautiful campuses to visit and ranks among the top 50 liberal arts colleges nation-wide each year.	Today, international leaders met with the President of the United States to discuss options for dealing with growing instability in global financial markets. President Obama indicated that the United States is cautiously optimistic about the potential for significant improvements in several struggling world economies pending the results of upcoming elections. The President indicated that the United States will take whatever actions necessary to promote continued stability in the global financial markets.

一般情况下，若主题在文档中出现多次，则被认为该文档与特定主题（查询词项）更相关。

这是 Solr 默认相关度公式中  $tf$  的基本前提。查询词项在某一文档中出现次数越多，则该文档被视为越相关。但是，某一词项在文档中的出现次数为 10，并不意味着该文档的相关度就提升 10 倍，因此我们会对文档中查询词项出现的次数开平方根来计算  $tf$ ，以此减少查询词项多次出现对相关度得分的额外加成。

3.2.3 反向文档频次

查询词项并非都是“生来平等”的。假设有人在图书馆目录中搜索 Dr. Seuss 的 *The Cat in the Hat*，那么位于搜索结果前列的文档将是包含了 *the* 和 *in* 这些高频词（而不是 *cat* 和 *hat*）的文档。然而，在查询匹配中较少见的词比常见词具有更好的区分度。

反向文档频次是查询词项罕见程度的度量，根据文档频次（含有该查询词项的总文档数）计算它的逆，参见图 3.7 的计算公式。

因为  $idf$  表示词项同时出现在查询和文档中，因此在相关度公式计算中需要求平方。

图 3.8 展示了图书馆馆藏资源中 *the Cat in the Hat* 每个词的“罕见度”， $idf$  值越高，则对应的词项越大。



图 3.8 根据  $idf$  值生成的词项相对显著性图示。词项越罕见，其图示越大，表示反向文档频次较大

同样，假如在一大堆简历中寻找一位经验丰富的 Solr 开发团队负责人的资料，不能用 `an`、`team` 或 `experienced` 这样的词汇来计算文档匹配度，而是要用图 3.9 中字号最大的重要词项。

an experienced **solr** development team lead

图 3.9 根据 idf 值生成的词项相对得分的另一示例。再一次表明，词项 idf 值越大表明其越罕见，与查询词项越相关，对它使用较大的字号表示

显然，用户正在寻找熟悉 Solr 并能领导团队的人，因此这些词在文档匹配中的权重更高。

词项频次与反向文档频次在相关度计算中起到了相互平衡作用。词项频次“奖励”了在一个文档中出现多次的词项，而反向文档频次“惩罚”了在多个文档中普遍出现的词项。因此，英语中的常见词，例如 `the`、`an` 和 `of` 等在任何文档中都会频繁出现的词汇，最终会拉低相关度得分。

### 3.2.4 词项权重

把相关度计算全部交给 Solr 来完成是没有必要的。如果你拥有专业领域知识，对内容本身的特定字段或词项的相对重要性能做出判断的话，那么就可以在索引阶段或查询阶段相应调整这些字段或词项的权重。

查询阶段权重设置，具有最灵活和简单的权重设置理解形式，使用如下语法。

- 查询：`title:(solr in action)^2.5 description:(solr in action)`

该示例将 `title` 字段的查询短语权重设置为 2.5，`description` 字段保留默认权重 1.0。（除非另有说明，否则词项就会被赋予默认权重 1.0。这意味着计算得分时乘以 1，或按初始计算方式。）

如果权重小于 1.0，查询权重也可用于“惩罚”特定词项。

- 查询：`title:(solr in action) description:(solr in action)^0.2`

需要注意，权重小于 1 仍然表示正向权重。这并不是在“惩罚”包含那些词项的文档，只是让该词项的重要性比那些默认权重的词项弱一些。

查询阶段的权重可应用在查询表达式的任何部分。

- 查询：`title:(solr^2 in^.01 action^1.5)^3 OR "solr in action" ^2.5`

特定查询解析器甚至可以默认对整个字段设置权重，第 7 章会详细介绍。

除了可以在查询阶段权重设置之外，在索引阶段也可以为文档字段设置权重。这些权重会被分解到字段规范中，下一节将具体介绍。

### 3.2.5 规范化因子

Solr 默认的相关度公式计算了三种规范化因子：字段规范、查询规范和协调因子。

#### 字段规范

字段规范 (field norm) 因子是以每个文档为基础的特定字段重要性的因子组合。字段规范在索引创建时进行计算，其表示为 Solr 索引中每个字段的一个附加字节。

该字节包含许多信息：文档被索引时的权重集、字段被索引时的权重集、惩罚长文档的同时提升短文档的长度归一化因子（前提是给定关键词在长文档中出现的可能性更大，因此相关性较低）。字段规范的公式计算如图 3.10 所示。

$$\text{norm}(t,d) = d.\text{getBoost}() \cdot \text{lengthNorm}(f) \cdot f.\text{getBoost}()$$

图 3.10 字段规范计算。字段规范由匹配文档的权重、匹配字段的权重及惩罚长文档的长度归一化因子组成。这三个完全独立的数据以单个字节存储在 Solr 索引中，这是组合为一个字段规范变量的唯一依据

`d.getBoost()` 分量表示发送至 Solr 的文档的权重，`f.getBoost()` 分量表示字段的权重。值得一提的是，Solr 允许同一字段被多次添加到文档（执行的背后是将字段的每个单独条目映射到同一个底层的 Lucene 字段）。由于重复的字段最终被映射到同一个底层字段，因此如果该字段存在多个副本，`f.getBoost()` 会成为多个相同名称字段的每个权重相乘的积。

如果 `title` 字段被添加到一个文档中三次，三次权重分别为 3、1 和 0.5，那么三个字段（即底层同一个字段）的 `f.getBoost()` 值为：

$$\bullet (3) \cdot (1) \cdot (0.5) = 1.5$$

除了索引阶段的权重设置之外，长度归一参数也被纳入到字段规范中。长度归一参数取值等于字段中词项数量的平方根。

值得一提的是，文档权重在内部执行时会被赋给该文档的每个字段作为权重。换句话说，设置文档权重与为该文档中每个字段赋予文档相同的权重值，这两种做法并没有差异。所有文档的权重最终存储在各自文档的每个字段规范中。

长度归一化的目的是调整不同长度的文档。通常，特定词项在长文档中出现次数可能较多，通过归一化可以消除较长文档的这一优势。

举例来说，当搜索关键词 Beijing 时，你更想要一篇 5 次提到 Beijing 的新闻报道，还是一本晦涩的 300 页图书，恰巧这本书中也出现了 5 次 Beijing？一般而言，当其他条件都相同时，这篇新闻报道会被认为是较好的匹配。这就是长度归一试图解决的问题。

总体的字段规范通过文档权重、字段权重和长度归一的乘积计算而来，被编码

为单字节并存储在 Solr 索引中。由于乘积编码的信息总量超过了单字节的容量，因此编码过程中会损失一些精度。现实中，这种保真度的损失对总体相关度的影响可以忽略不计，只有在所有其他相关度因素都不一致的情况下才会体现出较大差异。

### 查询规范

查询规范是 Solr 默认相关度计算中较少被关注的因子之一。由于同一个 queryNorm 应用于所有文档，因此它不影响总体的相关度排序，它仅用于查询之间进行比较时得分计算的规范化因子。该因子是每个查询词项的权重平方之和，再将它与相关度得分的其余部分进行相乘，从而实现规范化。查询规范不应影响与给定查询相匹配的每个文档的相对权重。

### 协调因子

协调因子是 Solr 默认相关度计算的最后一个规范化因子。它的作用是衡量每个文档匹配的查询数量，查询举例说明如下。

- 查询: Accountant AND ("San Francisco" OR "New York" OR "Paris")

你可能更倾向于找到每个城市的会计师及其所在办公室，而不是一遍遍提及“New York”的一位会计师。

若 4 个词项全部匹配到，则协调因子是 4/4；若匹配到其中 3 个词项，则协调因子是 3/4；若只匹配到 1 个词项，则协调因子是 1/4。

协调因子的理念是，所有的事物都是平等的，包含很多查询词项的文档应该比只包含几个查询词项的其他文档得分更高。

我们已经讨论了 Solr 默认相关度算法的主要组成部分。首先讨论了相关度得分计算最关键的两个部分 tf 和 idf。然后介绍了权重设置和规范化因子，对 tf 和 idf 计算的得分进行完善。在对相关度评分公式有了充分理解和掌握后，接下来讨论一下搜索系统返回结果集的总体质量评价的两个重要指标：查准率与查全率。

## 3.3 查准率与查全率

信息检索中的查准率 Precision(精确性的度量)与查全率 Recall(全面性的度量)<sup>2</sup>概念解释起来很简单，在构建搜索应用或解释返回的搜索结果为什么不能满足业务需求时，理解这两个概念非常重要。此处简要介绍这两个概念。

### 3.3.1 查准率

搜索结果集（与查询匹配的文档集）的查准率在试图回答这样一个问题：返回

2 译者注 在计算机领域，Precision 与 Recall 的另一种译法是“精确率”和“召回率”。

的这些文档是不是我想要寻找的？

查准率的具体定义如下（介于 0.0 和 1.0 之间）：

# 正确匹配的文档数量 / # 返回的文档数量

让我们回到 3.1 节中有关购房的图书搜索示例。根据自主标准判断，表 3.10 中所列的图书被视为较好地匹配了查询。

表 3.10 相关图书列表

相关图书
1 The Beginner's Guide to Buying a House
2 How to Buy Your First House
3 Purchasing a Home

在此示例中，其他所有图书都与购房不相关，表 3.11 列举了一些不相关图书。

表 3.11 不相关图书列表

不相关图书
4 A Fun Guide to Cooking
5 How to Raise a Child
6 Buying a New Car

本例中，假设应该返回的文档都返回了（文档 1、2 和 3），那么该查询的查准率为 1.0（3 个正确匹配 / 3 个全部匹配），这是理想情况。

但是，如果返回 6 个结果，那么查准率仅为 0.5，因为返回的结果中有一半是不正确的，即与查询无关。

同样地，如果只返回了 1 个相关结果（比如文档 2），查准率依然是 1.0，因为返回的结果都是正确的。可见，查准率是衡量结果与查询是否相关的一个指标，但是它并不关注全面性，即使只返回了 100 万个匹配文档中的 1 个，查准率依然会判定为最佳匹配。

由于查准率仅考虑了返回结果的整体精确性，而未考虑结果集的全面性，因此需要将查全率（全面性指标）与查准率搭配使用。

3.3.2 查全率

查准率衡量的是每个返回的搜索结果是否正确。查全率衡量的则是搜索结果的

全面性。查全率回答这样的问题：返回了多少正确的文档？

查全率的具体定义如下：

$\# \text{ 正确匹配的文档数量} / (\# \text{ 正确匹配的文档数} + \# \text{ 错误匹配的文档数})$

作为查全率的示例，我们将上一小节中相关图书和不相关图书的列表改造一下，如表 3.12 所示。

表 3.12 相关与不相关图书列表

相关图书	不相关图书
1 The Beginner's Guide to Buying a House	4 A Fun Guide to Cooking
2 How to Buy Your First House	5 How to Raise a Child
3 Purchasing a Home	6 Buying a New Car

如果查询返回了全部 6 个文档，则查全率为 1，因为找到了所有正确的文档，不存在遗漏。但这种情况下的查准率为 0.5。

如果仅返回一个文档，则查全率为 1/3，因为没有找到其他两个应该返回的文档。

查准率与查全率的重要区别是：如果返回的结果是正确的，则查准率高；如果正确的结果都被返回来了，则查全率高。查全率不关心返回的结果是否都是正确的，而查准率不关心正确的结果是否都被返回来了。

下一节讨论在查准率与查全率之间达到平衡的策略。

### 3.3.3 达到平衡

虽然查准率与查全率之间存在明显的互逆关系，但它们并不是相互排斥的。在前面的示例中，如果查询只返回文档 1、2 和 3，那么查准率与查全率就都为 1.0。这是因为所有正确的结果都被找到了。

最大限度提升查准率与查全率是绝大多数搜索相关度优化的最终目标。人为或手动调整结果集看似简单，但事实上，这是一个具有挑战性的难题。

Solr 采用的许多技术能够提高查准率与查全率，其中大多数技术更倾向于提高返回完整文档集的查全率。得力的文本分析（找到词汇的多个变体）是寻找更多匹配结果的一个典型示例。如果文本分析力度过大，匹配到了错误的词汇变体，那么这些额外的匹配结果可能会影响查准率。

Solr 中平衡查准率与查全率的一种常见方式是：在整个结果集上计算查全率，仅在搜索结果第一页（或少数页）上计算查准率。根据这一模型，调节 Solr 的相关度评分的计算方式，让更好的匹配结果被提升到搜索结果的顶部，而许多不良的匹配出现在搜索结果的底部。

这仅仅是一种问题解决方法。许多搜索网站可能想要显示尽可能多的内容。而



且这些网站知道访问者不会访问除了前几页之外的后续页面，因此他们在前几页显示更精确的结果，在后续页面显示精确度略低的匹配结果。这将得到高查全率，因为在整个结果集上“宽松地”对待能够匹配初始查询的那些关键词。与此同时，由于对搜索结果长列表顶端的最佳匹配进行了提升，前两页结果的查准率也很高。

如何做到查准率与查全率的最佳平衡，最终取决于现实情况。在法律方面的搜索中，查全率的重要性更高，因为如果遗漏了文档，就可能会导致一定的法律后果。对于其他现实需求而言，可能仅仅找到一些最佳匹配就可以了，若没有找到精确匹配查询词项的结果，不返回结果即可。

大多数搜索应用处在这两个极端之间。由于大多数情况下没有所谓正确答案，所以在查准率与查全率之间取得适度平衡成为一项长期的挑战。不管怎样，理解查准率与查全率以及如何在两者之间做出调整——使得杠杆向某一边倾斜（另一边上升），这是有效改进搜索结果质量的关键。第 16 章将专门介绍相关度，你会再次见到查准率与查全率之间的互逆关系。

## 3.4 搜索的规模化

除了速度、相关度与强大的文本搜索功能之外，Solr 最吸引人的方面之一是它的规模化。通过添加服务器，Solr 可以扩展到处理数十亿文档与无限查询请求数量。第 12 章和第 13 章将深入介绍生产环境下 Solr 的规模化，本节只介绍一些基础知识，主要涉及可扩展搜索引擎运行的必要特征。具体来说，我们将讨论以下内容：Solr 文档的非规范化本质、跨服务器线性扩展的理由、分布式搜索原理、从概念上理解服务器到服务器集群的转变，以及 Solr 扩展的一些局限。

### 3.4.1 非规范化文档

Solr 的核心概念是所有文档去除规范化。非规范化文档（denormalized document）指文档中的所有字段是自包含的，允许这些字段的值在多个文档中重复出现。对许多 NoSQL 技术来说，非规范化数据的概念很常见。非规范化的一个很好的示例是用户个人资料，包含 city、state 和 postalCode 字段。大多数情况下，对于每个特定的 postalCode 取值而言，所有文档中的 city 和 state 字段取值应该是相同的。这与规范化文档形成对比，文档各部分之间的关系可以拆分成多个更小的文档，在查询阶段各部分再连接起来。规范化文档只包含一个 postalCode 字段，每个特定的 postalCode 对应一个单独的位置文档，这样的话，city 和 state 就不需要在每个用户个人资料中重复出现了。如果你具备构建规范化表格和关系型数据库的知识和经验，那么在 Solr 的内容建模中请先放下这些知识和经验。图 3.11 展示了传统规范化数据库表的建模，大大的“×”说明它在 Solr 的数据建模



中明显不适用。

用户:

Id	UserName	About	Location	Company	LastModified
456	Coco	I'm a real monkey	1	1	2013-06-01 T15:26:37Z
123	John Doe	Senior Software Engineer with 10 years of experience with java, ruby, and .net	2	1	2013-06-05 T12:25:12Z

位置:

Id	City	State
1	Norcross	GA
2	Atlanta	GA
3	Decatur	GA

公司:

Id	CompanyName	CompanyDescription	Location
1	Code Monkeys R Us, LLC	we write lots of code	2

图 3.11 Solr 文档不遵循传统关系型数据库的规范化模型。该图说明了不要以规范化模型去考量 Solr 文档。Solr 不关注相互关联的多个实体，Solr 将文档数据建模为一个平面的、非规范化的数据结构，如代码清单 3.2 所示

请注意，图 3.11 中的信息代表了在同一家公司（Code Monkeys R Us, LLC.）任职的两个用户。如图中所示，数据被规范化后分别放入单独的表中，包括雇员个人信息、位置与公司三个字段。这不是 Solr 文档的表示方式，代码清单 3.2 将每个雇员信息去除规范化后映射为一个 Solr 文档。

### 代码清单 3.2 两个非规范化用户文档

```
<doc>
  <field name="id">123</field>
  <field name="username">John Doe</field>
  <field name="about">Senior Software Engineer with 10 years of
    experience with java, ruby, and .net
  </field>
  <field name="usercity">Atlanta</field>
  <field name="userstate">Georgia</field>
  <field name="companyname">Code Monkeys R Us, LLC</field>
  <field name="companydescription">we write lots of code</field>
  <field name="companycity">Decatur</field>
  <field name="companystate">Georgia</field>
  <field name="lastmodified">2013-06-05T12:25:12Z</field>
</doc>
```

① 第一位用户的公司信息。

```
<doc>
  <field name="id">456</field>
  <field name="username">Coco</field>
  <field name="about">I'm a real monkey</field>
  <field name="usercity">Norcross</field>
  <field name="userstate">Georgia</field>
  <field name="companyname">Code Monkeys R Us, LLC</field>
  <field name="companydescription">we write lots of code</field>
  <field name="companycity">Decatur</field>
  <field name="companystate">Georgia</field>
  <field name="lastmodified">2013-06-01T15:26:37Z</field>
</doc>
```

2 第二位用户那里重复了相同的公司信息。

请注意，所有的公司信息在第一个用户文档和第二个用户文档中重复出现，这似乎有悖于规范化数据库设计中减少数据冗余与最小化数据依赖的原则。在传统关系型数据库中，可以通过连接多个表的数据来实现查询。虽然 Solr 支持基本的连接查询功能（第 15 章会介绍），但建议仅在不能进行内容去规范化的情况下使用该功能。Solr 能够识别出映射到文档的词汇，但并不能识别文档之间的关系。也就是说，如果你想要搜索 Decatur,GA 公司的员工，那么所有的用户个人资料需要包含 companycity 和 companystate 字段才能成功实现检索。

虽然非规范化文档数据模型看似存在一些局限，但它提供了极具可扩展性的规模化优势。在每个文档都是自包含的前提下，可以跨服务器对文档进行分区，而无需保持单一服务器上的相关文档（因为文档之间是相互独立的）。文档独立性的基本假设允许在多个文档分区和多台服务器之间并发查询，从而改进查询性能，最终能够实现并发处理数十亿文档的查询。Solr 跨分区与服务器的扩展能力称为分布式搜索，将在下一小节介绍。

### 3.4.2 分布式搜索

如果每一个重要的数据操作都能运行在单台服务器上，那么整个世界就变得简单了。然而，现实情况是，同时发出过多的查询请求，或者需要在单台服务器上处理太多的搜索数据，这些都会导致搜索服务器超载。

后一种情况下，有必要将内容拆分到两个单独的 Solr 索引中，每一个索引包含单独的一部分数据。每次搜索运行时，查询会被同时发送到两台服务器上，分别进行处理后汇总在一起再返回给搜索引擎。

Solr 的分布式搜索功能开箱即用。第 12 章在讨论生产环境下的 Solr 规模化，介绍如何手工将数据分段到多个分区上。从理论上讲，每个 Solr 索引（称为 Solr 的内核）使用它的唯一 URL 来调用，并使用以下语法实现多个 Solr 内核的聚合搜索：

```
http://box1:8983/solr/core1/select?q=*&shards=box1:8983/solr/core1,  
box2:8983/solr/core2,box2:8983/solr/core3
```

上述例子具有如下 4 个特点：

- 该分片参数用于指定一个或多个 Solr 内核的地址。分片是索引的一个分区，URL 的分片参数告诉 Solr，对不同 Solr 内核上发现的多分区数据进行结果聚合。
- 在 box1 和 core1 上搜索的 Solr 内核也包含在分片列表中。除非发起明确的请求（如前所示），否则内核不会自动搜索。
- 分布式搜索会对多个服务器进行搜索。
- 不要求将独立的 Solr 内核放在单独的机器上。它们可以在同一台机器上，例如，这里的示例内核 core2 和 core3 都放在 box2 机器上。

根据 Solr 扩展属性开展搜索很重要。因为多个 Solr 内核的分布式搜索运行在每个索引分区上，所以 Solr 理论上应该可以进行线性扩展。如果将一个 Solr 索引拆分为文档数量相同的两个索引，那么减去结果聚合开销的话，跨两个索引的分布式搜索能够提速接近 50%。

理论上讲，服务器可以增加任意数量（现实中服务器数量总有上限）。增加一个索引分区（假设文档总数相同）时，总查询速度的理论公式如下：

$(N+1 \text{ 个索引的查询速度}) = \text{结果聚合的开销} + (N \text{ 个索引的查询速度}) / (N+1)$

这个公式可用于对数据平均分割的分区数量增加所带来的好处进行估算。由于 Solr 是近似线性扩展的，若不考虑服务器资源负载过重的限制，可以通过添加 Solr 内核（分区）数量来减少查询时间。

### 3.4.3 集群 vs. 服务器

上一节介绍了分布式搜索的概念，可以用它来处理大文档集合的扩展问题。通过向系统中添加多个同样的服务器也可以平衡高查询量的负载。

这两种策略取决于服务器与集群的观念转变。服务器集群的定义为协同工作以执行统一功能的多台服务器。

如下示例与 3.4.2 节的示例看起来有点类似：

```
http://box1:8983/solr/core1/select?q=*&shards=box1:8983/solr/core1,  
box2:8983/solr/core2
```

该示例在两个 Solr 内核（box1 上的 core1 和 box2 上的 core2）上执行分布式搜索。如果在执行过程中 box2 发生故障了，在 box1 执行的查询会发生什么情况？

代码清单 3.3 显示了这种情况下 Solr 的响应，包括了 box2 连接失败的错误消息。

代码清单 3.3 分布式搜索出错举例 (RemoteServer, 远程服务器故障)

```
<response>
  <lst name="responseHeader">
    <int name="status">500</int>
    <int name="QTime">1076</int>
    <lst name="params">
      <str name="shards">
        box1:8983/solr/core1,
        box2:8983/solr/core2
      </str>
    </lst>
  </lst>
  <lst name="error">
    <str name="msg">
      org.apache.solr.client.solrj.SolrServerException: IOException
      occurred when talking to server at: http://box2:8983/solr/core2
    </str>
    <str name="trace">...</str>
    <int name="code">500</int>
  </lst>
</response>
```

请注意, 这种情况下的服务器是相互依存的。如果其中一台无法被搜索, 它们就都不能被搜索, 导致整体出错, 如代码清单 3.3 中的异常所示。因此, 构建 Solr 解决方案时要考虑扩展性这一重要方面, 不要把所有鸡蛋放在一个篮子里, 要采用服务器集群取代单一服务器, 由这些服务器组成一个计算资源来提供服务。Solr 内置的 Apache ZooKeeper 能够提供优秀的集群管理功能, 第 13 章会具体介绍。

以上讨论了搜索规模化背后的关键概念, 接下来我们需要对 Solr 的局限性有所了解, 下一节将讨论这一话题。

### 3.4.4 Solr 的局限

Solr 是强大的基于文档的 NoSQL 数据存储方案, 支持全文搜索与数据分析。前面讨论了 Solr 的倒排索引与基于关键词的复杂布尔查询功能所带来的有力优势, 还讲解了相关度的重要性, 我们还看到 Solr 能够提供跨多台服务器, 进行或多或少的限行扩展, 从而对额外的内容和查询量进行处理。那么, Solr 对哪些情况而言不是一个好的解决方案呢? Solr 的局限有哪些?

之前已经了解了 Solr 的一个局限, 即 Solr 在文档处理上不以任何方式构成关系型。它不适用于在不同文档的不同字段上连接大量数据, 也无法在多台服务器上执行 join 操作。与关系型数据库相比, 这是 Solr 的基本局限。由于克服了关系型数据库的扩展局限性, 文档独立性假设是许多 NoSQL 技术常用的折衷方式。

之前也讨论了 Solr 文档的非规范化属性: 数据是冗余的, 每个文档中相同字段

的数据会重复出现。当多个文档共享的某个字段的数据发生变化时，这会带来很麻烦的问题。

举例来说，假设要构建一个社交网络用户个人资料的搜索引擎。其中一个用户 John Doe 与另一个用户 Coco 成为朋友。这不仅需要更新 John 和 Coco 的用户信息，还要更新 John 和 Coco 所有朋友的“二级连接”字段。对“两个用户成为朋友”这一基本操作就需要更新上百个文档。这一点就源于 Solr 不以任何方式构成关系型。

Solr 的另一个局限是，它目前主要用作文档存储方案，也就是说，你可以插入、删除与更新文档，但不能对字段轻松地做这些操作。当前，Solr 对单个字段仅能做一些简单的处理，而且前提是字段的原始值存储在索引中，但这样做又比较浪费。即便如此，Solr 还是基于内部存储字段，通过重做索引来更新整个文档。这就意味着，每当新的字段添加到 Solr 或者已有字段的内容发生变化时，Solr 索引的每个文档必须在搜索文档的新字段填充数据之前全部重新处理。许多其他的 NoSQL 系统也存在同样的问题，但值得注意的是，目前在语料库中添加或修改所有文档中的字段时，需要非常少量的文档更新协调，确保 Solr 能够及时完成更新。

Solr 也适用于一种特殊情况：处理少量搜索词的查询，快速检索每个搜索词匹配的文档，计算相关度得分，对其进行排序并仅显示少量的返回结果。Solr 不适用于处理非常长的查询（包含上千个搜索词的查询）或给用户返回大量搜索结果集。

Solr 的最后一个值得一提的局限是它的可扩展性（自动添加和删除服务器，以及处理负载而重新分发内容的能力）不够灵活。虽然 Solr 可以很好地跨服务器进行扩展，但在全自动扩展下，它本身的灵活性还不够。SolrCloud 最新版本中使用 Apache ZooKeeper 来管理集群（参见第 13 章）。这是一个不错的开端，不过仍然还有许多功能尚待研究。例如，Solr 尚不能自动重新分片内容和增加索引复本的数量，从而动态处理内容增长和查询负载。社区中许多聪明人正为这方面的目标长期努力着，Solr 的各个新版本也在越来越靠近这些目标。

## 3.5 本章小结

本章介绍了 Solr 搜索能力的重要基础理论知识，讨论了 Solr 倒排索引的结构，以及它是如何将词项映射到文档，并快速执行复杂布尔查询的。本章还讨论了模糊查询和短语查询，了解了它们是如何使用位置信息在 Solr 的索引中匹配词项与短语的拼写错误及变种形式。

本章深度探讨了 Solr 的相关度，介绍了 Solr 的默认相关度公式，从理论上解释了相关度评分中每一部分存在的理由和作用。

然后简要介绍了查准率与查全率的概念。这是信息检索领域中两个互相制约的评价指标，为判断搜索结果是否满足目标，提供了概念框架。

最后，我们讨论了 Solr 如何进行扩展，包括文档中内容去除规范化，以及当内容增长超出了单台机器的可处理能力时，利用分布式搜索确保查询并行化执行，以维持或减少搜索时间。之后，讨论了搜索架构扩展中的集群与服务器，介绍了 Solr 的一些局限和 Solr 不适用的情况。

至此，你应该已经具备了理解书中其余章节涉及的 Solr 核心功能的必要理论知识，牢固掌握了构建一个广受好评的搜索应用所需的重要概念。下一章将深入讨论 Solr 的关键配置设定，进一步探讨本章的许多功能。

# 配置Solr

## 本章要点

- 处理查询请求
- 利用搜索组件扩展查询
- 管理和预热搜索器
- 管理缓存

到目前为止，本书已经介绍了关于 Solr 的很多知识，但还没有涉及 Solr 背后的工作原理。本章将开始深入 Solr 的内部，介绍 Solr 的配置以及这些配置对 Solr 运行方式和性能的影响。Solr 的配置文件可能看上去十分庞杂，这是因为示例服务器中的配置文件涵盖了 Solr 几乎所有的配置内容。不过本章仅仅介绍 Solr 中最重要的配置，尤其是那些影响 Solr 处理客户端请求方式的配置。本章学到的知识将贯穿书中其他章节。阅读完本章，你将对查询语句在 Solr 服务器中的执行过程有深入的理解。

第 2 章曾讲过 Solr 是开箱即用的，即无须对原配置做任何修改就可使用。但在某些时候，需要根据特定的搜索需求对它进行优化。一般来说，大部分与 Solr 有关的配置都是与下面三个 XML 文件打交道：

- solr.xml——定义管理、日志、分片和 SolrCloud 的有关属性。
- solrconfig.xml——定义 Solr 内核的主要配置。
- schema.xml——定义索引结构，包括字段及其数据类型。

本章重点介绍 solrconfig.xml 文件。schema.xml 文件将在第 5 章介绍，它决定了索引的结构。至于 solr.xml，现在不需要对它做任何修改，我们将在第 12 章中与内



核管理 API 一起介绍它。

因为 Solr 的大部分配置文件都是 XML 格式, 所以本章包含大量 `solrconfig.xml` 文件中的 XML 代码片段。本章的重点并不是介绍具体的 XML 语法, 而是这些配置代码背后的原理, 况且这些 XML 语法都是自解释型的。

首先, 我们从配置的角度来学习 Solr 服务器启动的过程。回想一下, 第 2 章曾提到, Solr 作为一个 Java Web 应用运行在 Jetty 中, 使用全局 Java 属性 (`solr.solr.home`) 来定位配置文件的根目录。以示例服务器为例, Solr 主目录 (`solr.solr.home`) 为 `$SOLR_INSTALL/example/solr/`。

接下来, Solr 在主目录下扫描包含 `core.properties` 文件的子文件夹, 该文件定义了 Solr 中自动发现内核的基本属性。例如, 在示例服务器中, `example/solr/collection1` 目录下有 `core.properties` 文件, 该文件中仅有一行代码 `name=collection1`, 定义了内核的名称, Solr 通过这行代码就能自动发现名为 `collection1` 的 Solr 内核。图 4.1 描述了 `core.properties` 文件和 `solrconfig.xml` 文件在 Solr 的初始化过程中是如何创建和设置 `collection1` 内核的。

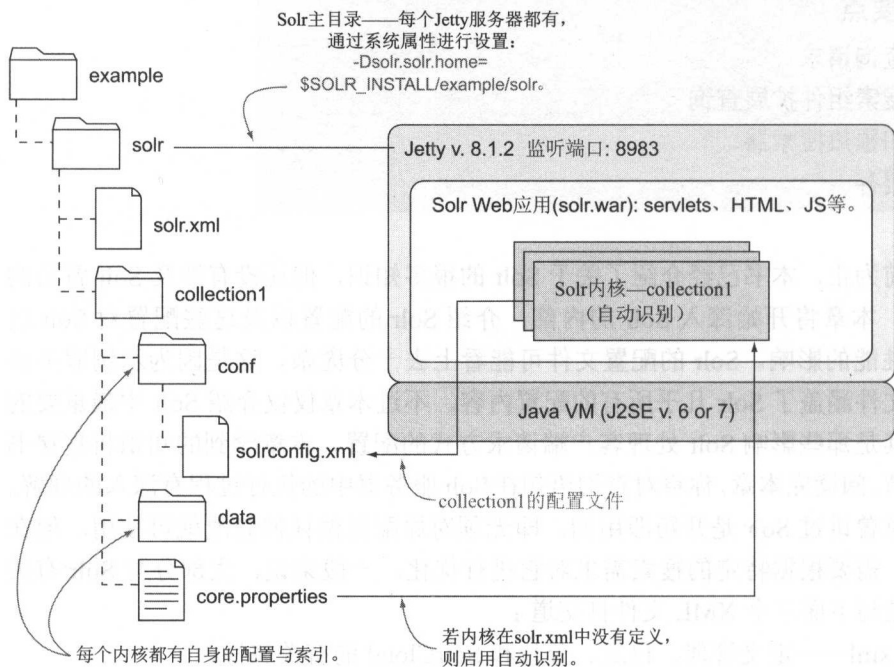


图 4.1 Solr 利用 `core.properties` 文件自动发现 `collection1`, 并在服务器初始化过程中利用 `solrconfig.xml` 文件对它进行配置

在 Solr 的早期版本里, 开发者需要在 `solr.xml` 文件中定义内核, 这样做的缺点

是需要先为内核创建专门的文件夹，再在 `solr.xml` 文件中添加该内核的定义。而新版本中利用 `core.properties` 文件省略了上述步骤，更重要的是，这种方式使得每个内核都是独立自包含的，不再需要一份中心配置文件来定义所有的内核。

示例服务器中 `collection1` 的配置文件 `core.properties` 仅仅定义了必要的内核名称。除此以外，还可以设置一系列可选参数对内核的定义做适当的调整。表 4.1 列举了在 `core.properties` 文件中可定义的参数。

表 4.1 `core.properties` 文件中内核自动发现的配置参数

参数	说明
<code>name</code>	内核；必备
<code>config</code>	指定配置文件的名称；默认为 <code>solrconfig.xml</code>
<code>dataDir</code>	为包含索引文件与更新日志（ <code>tlog</code> ）的目录指定路径；默认为实例目录下的 <code>data</code> 目录
<code>uLogDir</code>	为包含更新日志（ <code>tlog</code> ）的目录指定路径
<code>schema</code>	为模式文档命名；默认为 <code>schema.xml</code>
<code>shard</code>	设置该内核的分片 ID；更多分片信息参见第 12、13 章
<code>collection</code>	该内核属于的 SolrCloud 集合名称；第 13 章介绍集合
<code>loadOnStartup</code>	若为真，在 Solr 初始化过程中加载该内核，为该内核开启一个新的搜索器
<code>transient</code>	若达到 Solr 的 <code>transientCacheSize</code> 阈值（高级选项），该内核可以自动不加载

现阶段，你还不理解表 4.1 中列出的参数，因为本章的重点是学习 `collection1` 中的配置文件 `solrconfig.xml`。

需要重点理解的是，Solr 可以在启动期间利用 `core.properties` 文件自动发现内核。一旦内核被发现，Solr 就能定位到位于 `$SOLR_HOME/$instanceDir/conf/solrconfig.xml` 文件夹下的 `solrconfig.xml` 文件，其中 `core.properties` 文件位于 `$instanceDir/` 文件夹中。Solr 利用 `solrconfig.xml` 文件来初始化内核。

截至目前，本章介绍了 Solr 在启动期间定位配置文件的方式，接下来将介绍 `solrconfig.xml` 文件的主要部分，同时也是本章余下部分的重点。

## 4.1 solrconfig.xml文件概览

为方便解释 solrconfig.xml 文件中的一些概念, 本章将利用第 2 章中预先配置的示例服务器和 Solr 的示例搜索界面。建议读者在命令行中输入代码清单 4.1 中的命令, 启动第 2 章中的示例服务器。

### 代码清单 4.1 通过命令行启动示例服务器

```
cd $SOLR_INSTALL/example
java -jar start.jar
```

一旦成功地启动了服务器, 就可以在浏览器中访问 <http://localhost:8983/solr>, 进入 Solr 的管理控制台, 点击页面左边的 collection1 链接, 再点击 Files 链接, 将会看到 collection1 的所有配置文件的目录结构。点击 solrconfig.xml 文件链接, 可以看到正在运行的 collection1 中已启用的配置。代码清单 4.2 展示了一个缩略版的 solrconfig.xml 文件, 以便于读者了解该配置文件的主要内容。

### 代码清单 4.2 缩略版 solrconfig.xml

Lib 目录 存放 Solr 的扩展 JAR 文 件 (参 见 4.1.3 节)。	<pre> &lt;config&gt;   &lt;.luceneMatchVersion&gt;4.7&lt;/luceneMatchVersion&gt;   &lt;lib dir="../../contrib/extraction/lib" regex=".*\.jar" /&gt;   &lt;dataDir&gt;\${solr.data.dir:}&lt;/dataDir&gt;   &lt;directoryFactory name="DirectoryFactory" class="..." /&gt;   &lt;indexConfig&gt; ... &lt;/indexConfig&gt;   &lt;jmx /&gt;   &lt;updateHandler class="solr.DirectUpdateHandler2"&gt;     &lt;updateLog&gt; ... &lt;/updateLog&gt;     &lt;autoCommit&gt; ... &lt;/autoCommit&gt;   &lt;/updateHandler&gt;   &lt;query&gt;     &lt;filterCache ... /&gt;     &lt;queryResultCache ... /&gt;     &lt;documentCache ... /&gt;     &lt;listener event="newSearcher" class="solr.QuerySenderListener"&gt;       &lt;arr name="queries"&gt; ... &lt;/arr&gt;     &lt;/listener&gt;     &lt;listener event="firstSearcher" class="solr.QuerySenderListener"&gt;       &lt;arr name="queries"&gt; ... &lt;/arr&gt;     &lt;/listener&gt;   &lt;/query&gt; </pre>	<p>Lucene 激活版本相关功能 (参见 4.1.3 节)。</p>	<p>索引管理配置 (参见第 5 章)。</p>	<p>索引文档的更新处理器 (参见第 5 章)。</p>	<p>启用 Solr Mbeans 的 JMX 工具。</p>
	<p>搜索器事件的事件注册处理器, 例如, 预热新搜索器的查询执行 (参见 4.3 节)。</p>	<p>缓存管理设置。</p>			

```

<requestDispatcher handleSelect="false" >
  <requestParsers ... />
  <httpCaching never304="true" />
</requestDispatcher>
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults"> ... </lst>
  <lst name="appends"> ... </lst>
  <lst name="invariants"> ... </lst>
  <arr name="components"> ... </arr>
  <arr name="last-components"> ... </arr>
</requestHandler>
<searchComponent name="spellcheck"
  class="solr.SpellCheckComponent"> ... </searchComponent>
<updateRequestProcessorChain name="langid"> ...
  </updateRequestProcessorChain>
<queryResponseWriter name="json"
  class="solr.JSONResponseWriter"> ... </queryResponseWriter>
<valueSourceParser name="myfunc" ... />
<transformer name="db"
  class="com.mycompany.LoadFromDatabaseTransformer">
  ...
</transformer>
</config>

```

使用更新 - 请求处理器扩展索引行为, 例如, 语种检测。

统一请求分发器 (参见 4.2.1 节)。

请求处理器使用搜索组件链处理查询 (参见 4.2.4 节)。

查询拼写纠正的搜索组件示例。

为提升排名或排序文档, 声明一个自定义函数。

对搜索结果文档进行转换。

搜索结果格式为 JSON。

从代码清单 4.2 中可以看出, solrconfig.xml 文件由大量复杂的 XML 代码块组成。当然, 并不需要现在就掌握该配置文件的所有内容, 在碰到实际需求时再做具体了解即可。不过读者应该在脑海中对这些配置元素留下些许印象, 了解可以控制和扩展 Solr 中的哪些功能。从这里可以看出, Solr 的配置有很大的灵活性。

本章根据 Solr 配置之间的依赖关系来安排 Solr 配置的介绍顺序, 而不是根据配置元素在 XML 文档中的排列顺序依次介绍。例如, 尽管在 solrconfig.xml 文件中, 缓存的配置元素在请求处理的配置元素之前, 但本章仍在缓存之前介绍 Solr 的请求处理框架, 在利用缓存优化具体的客户端请求之前, 你需要理解这些请求在 Solr 中的处理方式。同时这也意味着在学习本章时, 读者将不得不来回翻阅配置文件。

## 第 5 章会介绍索引配置

solrconfig.xml 文件中包含了索引管理的配置, 但是索引相关的配置将在下一章中介绍, 这样读者对索引处理有了基本了解之后再学习它的配置。在学习第 5 章之前, 请忽略配置文件中的以下 XML 元素。

```

<dataDir> ... </dataDir>
<directoryFactory name="DirectoryFactory" class="...">
</directoryFactory>
<indexConfig> ... </indexConfig>
<updateHandler class="solr.DirectUpdateHandler2"> ...
</updateHandler>
<updateRequestProcessorChain name="langid"> ...
</updateRequestProcessorChain>

```

Lucene 和 Solr 具有良好向后兼容能力。<uceneMatchVersion> 元素

4.1.1 常见的 XML 数据结构和数据类型元素

如果你浏览一遍 solrconfig.xml 文件，就会看到各种用于表示数据结构和数据类型的 XML 元素。表 4.2 列举了 solrconfig.xml 文件中主要的数据类型元素，以及它们的简要说明和举例。这些元素还将出现在 XML 格式的搜索结果中，所以需要花一点时间熟悉这种 Solr 特有的语法。

表 4.2 Solr 中描述数据结构和数据类型的 XML 元素

元素	说明	举例
<arr>	命名的对象有序数组	<pre>&lt;arr name="last-components"&gt;   &lt;str&gt;spellcheck&lt;/str&gt; &lt;/arr&gt;</pre>
<lst>	命名的键值对有序列表	<pre>&lt;lst name="defaults"&gt;   &lt;str name="omitHeader"&gt;true&lt;/str&gt;   &lt;str name="wt"&gt;json&lt;/str&gt; &lt;/lst&gt;</pre>
<bool>	布尔值 true 或 false	<pre>&lt;bool&gt;true&lt;/bool&gt;</pre>
<str>	字符串值	<pre>&lt;str&gt;spellcheck&lt;/str&gt; 或 &lt;str name="wt"&gt;json&lt;/str&gt;</pre>
<int>	整数	<pre>&lt;int&gt;512&lt;/int&gt;</pre>
<long>	长整数	<pre>&lt;long&gt;1359936000000&lt;/long&gt;</pre>
<float>	单精度浮点值	<pre>&lt;float&gt;3.14&lt;/float&gt;</pre>
<double>	双精度浮点值	<pre>&lt;double&gt;3.14159265359&lt;/double&gt;</pre>

<arr> 和 <lst> 之间的最大不同是 <lst> 中的每个子元素都有一个 name 属性，而 <arr> 的子元素则没有。

4.1.2 配置文件更新的应用

Solr 配置的学习过程可能比较枯燥，为了在这个过程中保持兴趣，建议读者在阅读本章的同时动手做一些配置实验。请注意所做的修改只有在重新加载内核之后才会生效，Solr 并不会自动识别配置文件的更改，必须明确地告知 Solr 应用修改后的配置。目前，最简单的方法就是点击管理控制台中内核管理（Core Admin）页面的 Reload 按钮，如图 4.2 所示。

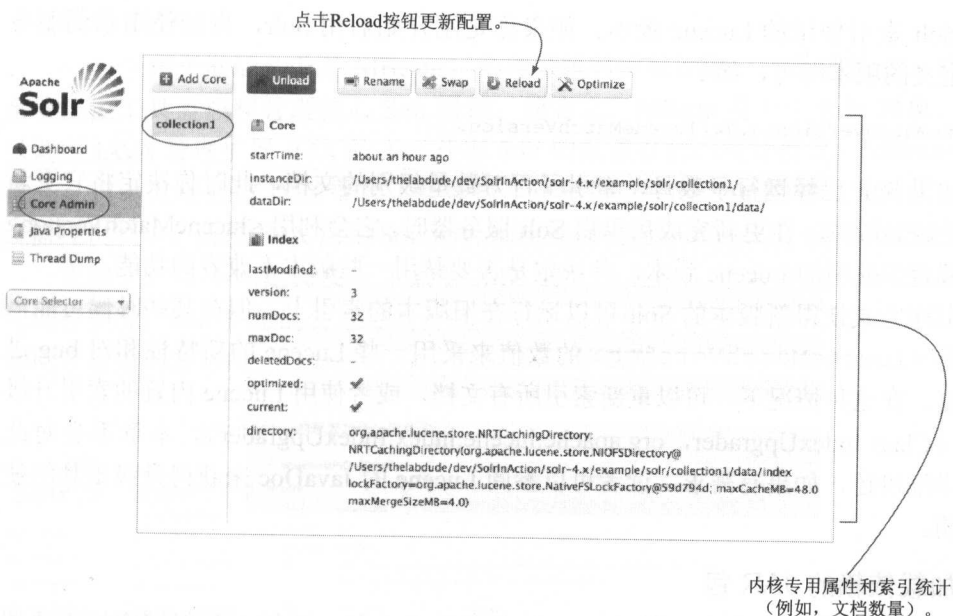


图 4.2 在内核管理页面重载内核以应用修改后的配置

如果 Solr 运行在本地, 不妨点击一下 collection1 的 Reload 按钮, 以确认是否可以应用修改后的配置文件。本章的最后将介绍另一种重载内核的方法, 即利用内核管理 API 编写程序来重载内核。

### 4.1.3 Solr 的其他配置

以上介绍了 Solr 配置的一些背景知识, 接下来将从 Solr 服务器的各种配置开始学习 solrconfig.xml 配置文件。代码清单 4.3 囊括了本小节要介绍的所有配置内容。

代码清单 4.3 solrconfig.xml 文件头部的全局配置

```
<config>
  < luceneMatchVersion>4.7</ luceneMatchVersion>
  < lib dir="../../contrib/langid/lib/" regex=".*\.jar" />
  < lib dir="../../dist/" regex="solr-langid-\d.*\.jar" />
  ...
  <jmx />
  ...
</config>
```

指明 Lucene 版本。

载入相关 JAR 包。

启用 JMX。

#### Lucene 版本

Lucene 和 Solr 具有良好的向后兼容能力。< luceneMatchVersion> 元素定

义了 Solr 索引使用的 Lucene 版本。如果你是刚开始启用 Solr，直接使用示例服务器中定义的版本即可，如：

```
<.luceneMatchVersion>4.7</.luceneMatchVersion>
```

如果 Solr 已经运行了数月，索引了百万数量级别的文档，此时你决定将它更新到一个较新版本。在更新完成后重启 Solr 服务器时，它会利用 <.luceneMatchVersion> 来了解索引使用的 Lucene 版本，并决定是否要禁用一些版本不兼容的功能。

上述方式使得新版本的 Solr 可以运行在旧版本的索引上。但在某些时候可能需要提升 <.luceneMatchVersion> 的数值来采用一些 Lucene 的新特性和对 bug 进行修复。在这种情况下，可以重新索引所有文档，或者使用 Lucene 内置的索引升级工具（Class IndexUpgrader, org.apache.lucene.index.IndexUpgrader）。本章不会对此内容详细讲述，如果有需求，读者可以参阅 Lucene 的 JavaDoc 来获得升级工具的使用说明。

### 加载依赖的 JAR 包

<lib> 元素允许向 Solr 的 classpath 中添加 JAR 包，以便 Solr 能够定位需要调用的插件类。下面来看 solrconfig.xml 文件中 <lib> 元素的工作原理。

```
<lib dir="../../../contrib/langid/lib/" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-langid-\d.*\.jar" />
```

每一个 <lib> 都指定了一个路径名和一个用于匹配 JAR 包的正则表达式。注意 dir 属性值是以内核的根目录为起点的相对路径，该根目录也常被称为内核的 instanceDir。对于示例服务器中的 collection1 而言，instanceDir 为 \$SOLR\_INSTALL/example/solr/collection1/；需要注意，\$SOLR\_INSTALL/ 是一个变量，它的值是 Solr 分发包解压缩时所处的绝对路径。因此，上述示例中的两个 <lib> 将下列 JAR 包添加到了 Solr 的 classpath 中：

- jsonic-1.2.7.jar（位于 contrib/langid/lib/）。
- langdetect-1.1-20120112.jar（位于 contrib/langid/lib/）。
- apache-solr-langid-4.7.0.jar（位于 dist/）。

需要注意的是 apache-solr-langid.jar 文件的版本号会随着 Solr 4 具体版本的不同而改变。另外，也可以使用 path 属性引用单个 JAR 包，如：

```
<lib path="../../../dist/solr-langid-4.7.0.jar" />
```

还可以将依赖的 JAR 包都放在 \$SOLR\_HOME/lib/ 目录中，如 \$SOLR\_INSTALL/example/solr/lib/。



## 启用 JMX

<jmx> 用于激活 Solr 的 MBeans，允许系统管理员使用一些类似 Nagios 的常用系统监控工具监控和管理核心 Solr 组件。简言之，MBean 是一个 Java 对象，它可以调用 JAVA 管理扩展 JMX 的 API 获得 Solr 的配置参数和统计信息。MBeans 可以被 Solr 自动发现并被 JMX 兼容工具监控。这样可以将对 Solr 监控集成到已有的系统监控机制中。通过 JMX 启用 Solr 的外部监控机制的方式将在第 12 章中详细介绍。

实际上，并不一定需要一个兼容 JMX 的外部监控工具来查看 Solr 的 MBeans 的运行实况。Solr 自带的管理控制台就支持对 Solr 中所有 MBean 的访问。图 4.3 是利用管理控制台访问 collection1 的 MBean 的截图。



在集合1中的插件/统计页访问Solr MBeans。

图 4.3 通过 collection1 的 Plugins/Stats 页监控 SolrCore MBean

本章后续部分将会介绍更多利用管理员控制台监控 Solr 中 MBean 的例子。接下来介绍 Solr 请求处理框架。

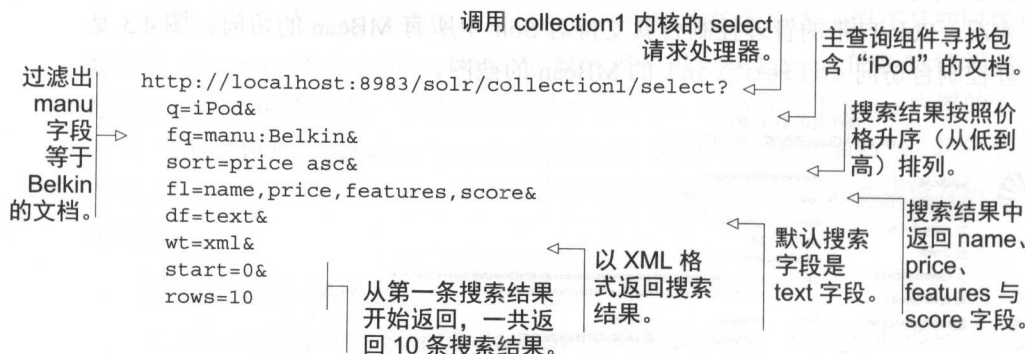
## 4.2 查询请求处理

Solr 的主要功能是搜索，所以搜索请求的处理是 Solr 最重要的处理之一。本节介绍 Solr 搜索请求处理框架，以及如何让自定义请求处理器更适合特殊搜索请求的操作方法。

## 4.2.1 请求处理简介

对 Solr 发起请求需要通过 HTTP。如果是查询请求，则为 HTTP GET 方法。如果是索引请求，则为 HTTP POST 方法。代码清单 4.4 展示了一个利用 HTTP GET 方法对示例 Solr 服务器发起查询的示例（同代码清单 2.1）。

代码清单 4.4 利用 HTTP GET 方法对示例 Solr 服务器发起查询请求



将代码清单中的 URL 输入到浏览器中，或者使用命令行工具 `curl`，或者利用本书附带的示例代码，这些方式都可以运行代码清单 4.4 中的示例。如果要使用本书的示例代码，只需在命令行中输入：

```
cd $SOLR_IN_ACTION
java -jar solr-in-action.jar listing #.#
```

注意需要将参数 `#.#` 替换为要执行的代码清单的编号，如 4.4。推荐使用本书附带示例代码中的 `http` 工具，这样可以省去额外的复制、粘贴或输入，并且该工具能够运行在所有 Java 平台上。当运行代码清单 4.4 中的示例后，输出如下：

```
java -jar solr-in-action.jar listing 4.4
INFO [main] (ExampleDriver.java:92) - Found example class sia.Listing for arg
http
INFO [main] (ExampleDriver.java:125) - Running example Listing with args: -
listing 4.4
Feb 13, 2013 6:21:32 PM org.apache.solr.client.solrj.impl.HttpClientUtil
createClient
INFO: Creating new http client,
config:maxConnections=128&maxConnectionsPerHost=32&followRedirects=false
Sending HTTP GET request (listing 4.4):
http://localhost:8983/solr/collection1/select?
q=iPod&
fq=manu:Belkin&
```

```

sort=price asc&
fl=name,price,features,score&
df=text&
wt=xml&
start=0&
rows=10

```

Solr returned: HTTP/1.1 200 OK

```

<?xml version="1.0" encoding="UTF-8"?>
<response>
...
</response>

```

示例代码的 http 工具提供了很多可选项，例如，允许修改 Solr 服务器的地址，或将返回数据格式从 XML 改为其他格式，如 JSON。要查看所有可选项，请在命令行中输入 `java-jar solr-in-action.jar listing -h`。

图 4.4 展示了 Solr 中处理客户端请求的事件队列和主要组件。从图 4.4 的左上角开始依次为：

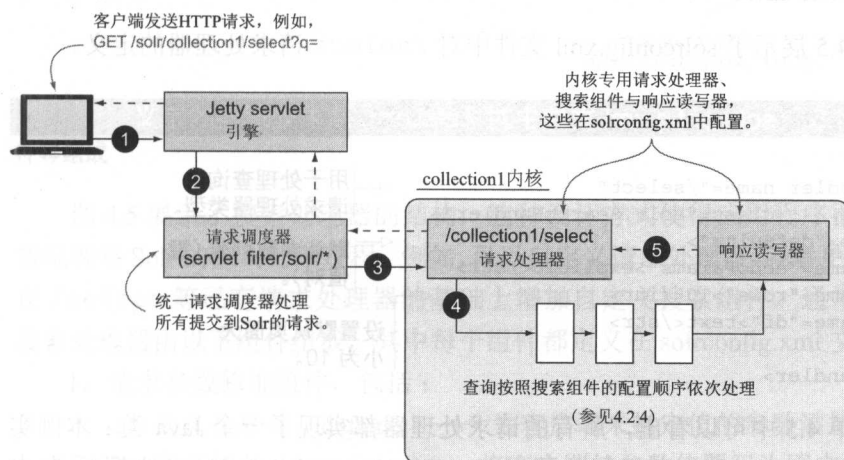


图 4.4 /select 请求处理器处理客户端请求的事件队列

1. 客户端应用将 HTTP GET 请求发送至 `http://localhost:8983/solr/collection1/select?q=...`，查询参数通过该 GET 请求中的查询字符串进行传递。
2. Jetty 接收客户端请求，并根据请求路径中 `/solr` 后的内容将该请求交给 Solr 中统一的请求分配器。从技术角度来说，统一请求分配器就是一个 Java servlet 过滤器，可以为 Solr Web 应用过滤出符合 `/*` 的 URL，详见 `org.apache.solr.servlet.SolrDispatchFilter` 类。
3. Solr 的请求分配器根据客户端请求路径中的 `collection1` 确定所查询的内核名称。接下来，请求分配器会定位到 `solrconfig.xml` 文件中定义的

/select 请求处理器。

4. /select 请求处理器利用一系列搜索组件（见 4.2.4 节）处理客户端请求。
5. 在处理完客户端请求之后，查询结果经由响应读写器组件进行格式化之后，返回给客户端应用；默认情况下，/select 请求处理器返回的结果数据为 XML 格式。有关响应读写器组件的更多介绍请参见 7.7 节。

请求分配器的主要职能是根据用户请求语句定位到处理该请求的内核，如 collection1，然后将用户请求交给该内核中已注册的对应请求处理器，在上例中为 /select。实际应用中，请求分配器的默认配置对于大多数应用来说已经足够了。

另外，开发者也可以自定义请求处理器或在 /select 等已有处理器的基础上做个性化的修改，这些都是很常见的处理方式。下一节将深入介绍 /select 请求处理器的工作原理，以便读者对自定义请求处理器有更深的体会。

## 4.2.2 搜索处理器

代码清单 4.5 展示了 solrconfig.xml 文件中对 /select 请求处理器的定义。

代码清单 4.5 solrconfig.xml 文件中对 /select 请求处理器的定义

```
Java 类执行请求处理器。
<requestHandler name="/select"
                 class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>
    <str name="df">text</str>
  </lst>
</requestHandler>
```

← 用于处理查询的请求处理器类型。

← 默认参数列表（键值对）。

← 设置默认页面大小为 10。

从代码清单 4.5 中可以看出，所有的请求处理器都实现了一个 Java 类：本例实现了 solr.SearchHandler。在运行时，solr.SearchHandler 被解析为内置的 Solr 类 org.apache.solr.handler.component.SearchHandler。一般来说，只要在 solrconfig.xml 文件中看到一个前缀为 solr. 的类，都应该想到是下列 Solr 的核心 Java 包之一："analysis."、"schema."、"handler."、"search."、"update."、"core."、"request."、"update.processor."、"util."、"spelling."、"handler.component." 或 "handler.dataimport."。在配置文件中利用 solr. 前缀简写 Solr 类的名称，运行时再解析至相应的内置 Solr 类，这种方式让配置文件看起来更加简洁有序。

Solr 中有两类主要的请求处理器：处理查询请求的搜索处理器和处理索引请求的更新处理器。下一章会在介绍索引的同时深入讲解更新处理器。本章重点介绍搜

索处理器处理查询请求的过程，参见图 4.5。

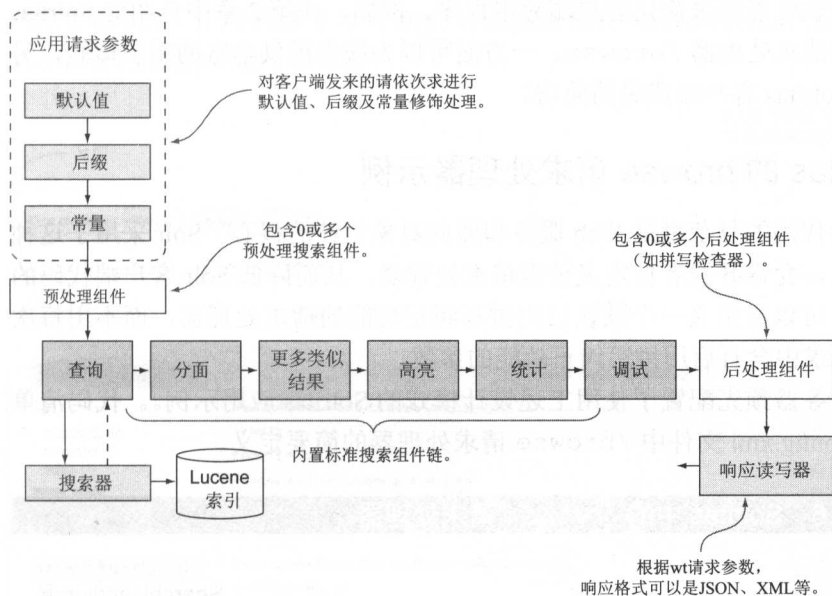


图 4.5 搜索请求处理器由参数修饰（默认、后缀和常量）、预处理组件、主搜索组件与后处理组件等组成

图 4.5 展示了搜索处理器的结构，这种设计方式使得应用程序很容易根据实际情况调整 Solr 的查询处理过程。例如，可以自定义请求处理器，或者更加常见的做法，在 `/select` 等已有请求处理器的基础上增加自定义搜索组件。通常情况下，一个搜索处理器由以下组件组成，其中每个组件都定义在 `solrconfig.xml` 文件中。

1. 请求参数修饰组件，包括：

- a. 默认值修饰（defaults）—— 为客户端未指定值的参数添加默认值。
- b. 常量修饰（invariants）—— 将客户端的参数值覆写为固定值。
- c. 后缀修饰（appends）—— 在客户端请求的末尾添加额外参数。

2. 预处理组件（first-components）—— 一组优先执行的可选搜索组件，执行预处理任务。

3. 主搜索组件（components）—— 一组链式组合的搜索组件，至少包含查询组件。

4. 后处理组件（last components）—— 一组可选的链式组合的搜索组件，执行后处理任务。

一般而言，一个请求处理器并不需要定义上述所有组件，比如代码清单 4.5 中，`/select` 请求处理器仅仅定义了默认值修饰组件。这意味着所有其他组件均默认为 `solr.SearchHandler` 的实现。如果请求处理器中的主组件未定义，则会使用一

组默认的搜索组件。这组默认搜索组件将在 4.2.4 节中介绍。在实际应用中，通常使用自定义的请求处理器来简化客户端应用程序。例如，在第 2 章中介绍的 Solritas 使用了自定义的请求处理器 /browse，一方面可以为搜索提供丰富的用户体验，另一方面保证了 Solritas 客户端代码的简洁。

### 4.2.3 Solritas 的 browse 请求处理器示例

降低客户端代码的复杂度是 Web 服务和面向对象设计的核心。Solr 采用了这种成熟的设计模式，允许开发者自定义搜索请求处理器，从而降低 Solr 客户端代码的复杂度。例如，可以自定义一个默认启用拼写纠正功能的请求处理器，而不用每次都要求客户端请求中含有启用拼写纠正功能的参数。

Solr 示例服务器预先配置了使用上述设计模式的 Solritas 应用示例。。代码清单 4.6 展示了 solrconfig.xml 文件中 /browse 请求处理器的简要定义。

代码清单 4.6 Solritas 中的 /browse 请求处理器

```

<requestHandler name="/browse" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <str name="wt">velocity</str>
    <str name="v.template">browse</str>
    <str name="v.layout">layout</str>
    <str name="title">Solritas</str>
    <str name="defType">edismax</str>
    <str name="qf">text^0.5 features^1.0 ...</str>
    <str name="mlt.qf">text^0.5 features^1.0 ...</str>
    <str name="facet">on</str>
    ...
    <str name="hl">on</str>
    ...
    <str name="spellcheck">on</str>
  </lst>
  <arr name="last-components">
    <str>spellcheck</str>
  </arr>
</requestHandler>

```

SearchHandler 调用查询处理管道。

查询参数默认列表。

VelocityResponseWriter 设置。

使用扩展的最大距离查询解析器。

更多类似结果组件的设置。

查询设置。

启用分面组件。

启用高亮组件。

启用拼写检查。

在管道处理的最后一步调用拼写检查组件。

建议读者将 solrconfig.xml 文件中所有关于 /browse 请求处理器的配置详细阅读一遍。从中，为了展示 Solr 中很多强大的特性，/browse 的这个实例在配置上花了很多精力。一开始使用 Solr 时，不需要做出像 /browse 那样复杂的配置，不过积累了一些 Solr 开发经验之后，可以尝试自定义一些 Solr 的请求处理器。

下面我们在 Solritas 实例中查看 /browse 请求处理器的运行情况。启动 Solr 示例服务器以后，在浏览器中访问 <http://localhost:8983/solr/collection1/browse>。如图 4.6 所示，在搜索框中输入 iPod。

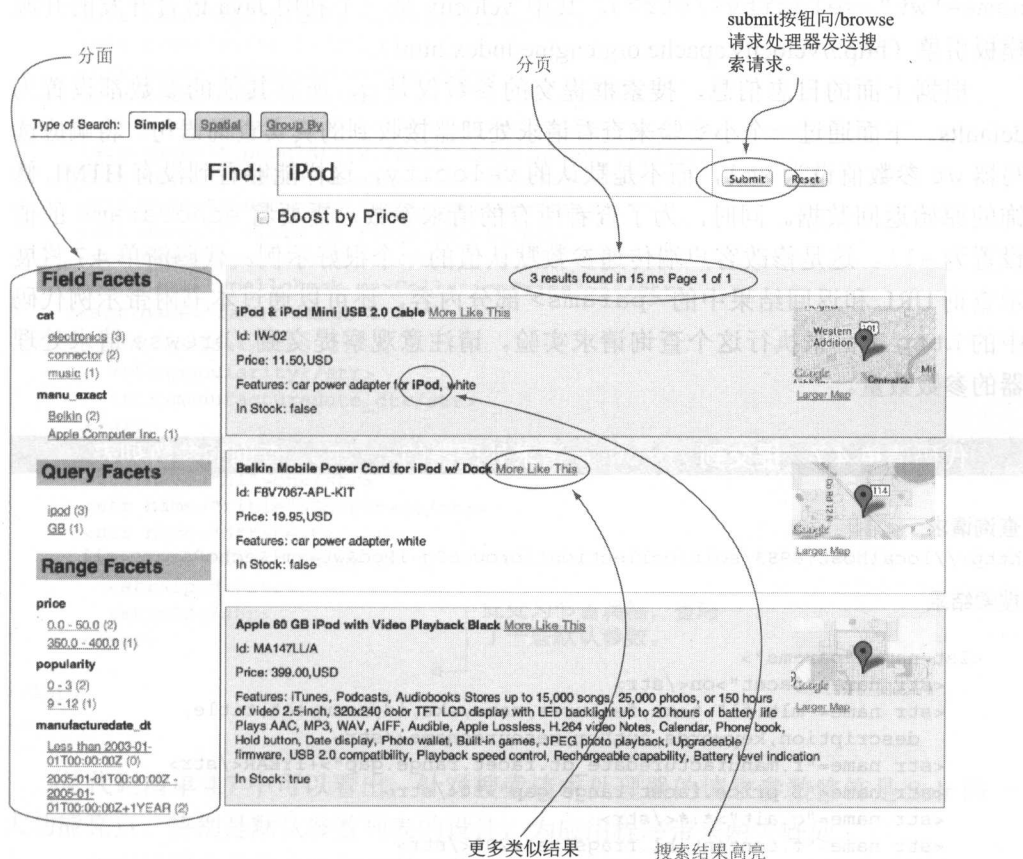


图 4.6 一个由 /browse 请求处理器驱动的 Solritas 示例程序截图

浏览图 4.6 中的信息，你会发现这条简单的查询语句激活了 Solr 中的许多搜索特性。在界面背后，Solritas 的搜索框向 /browse 请求处理器提交了一个查询。在日志文件中，可以看到如下信息：

```
INFO: [collection1] webapp=/solr path=/browse params={q=iPod} hits=3 status=0
      QTime=22
```

请注意，我们仅仅通过搜索框提交了一个参数 `q=iPod`，返回结果却包含了分面、更多类似结果、拼写检查、分页和高亮显示等功能。

这个简单的 `q=iPod` 查询充分体现了 Solr 所具有的丰富功能，这些功能在 /browse 请求处理器中是使用默认参数启用的。



从代码清单 4.6 中可以看出, defaults<lst> 元素是由许多个名称/取值对组成的有序列表, 如果客户端程序没有明确指定查询参数, 这个有序列表就为查询参数提供默认值。例如, 默认的响应数据格式参数 wt 是 velocity (<str name="wt">velocity</str>), 其中 Velocity 是一个利用 Java 语言开发的开源模板引擎 (<http://velocity.apache.org/engine/index.html>)。

根据上面的日志信息, 搜索框提交的参数仅是 q, 所有其他的参数都设置为 defaults。下面通过一个小实验来查看请求处理器接收到的真实查询语句。将响应读写器 wt 参数值设为 xml, 而不是默认的 velocity, 这样能够看到没有 HTML 修饰的原始返回数据。同时, 为了查看所有的请求参数, 需要将 echoParams 的值设置为 all。这是修改客户端传递参数默认值的一个很好示例。代码清单 4.7 将展示查询 URL 和返回结果中的 <params> 部分内容, 还可以通过本书附带示例代码中的 http 工具来执行这个查询请求实验。请注意观察提交到 /browse 请求处理器的参数数量。

#### 代码清单 4.7 提交到 /browse 请求处理器的 q=iPod 查询的参数列表

##### 查询请求

<http://localhost:8983/solr/collection1/browse?q=iPod&wt=xml&echoParams=all>

##### 搜索结果

```
...
<lst name="params">
  <str name="facet">on</str>
  <str name="mlt.fl">text,features,name,sku,id,manu,cat,title,
    description,keywords,author,resourceName</str>
  <str name="f.manufacturedate_dt.facet.range.gap">+1YEAR</str>
  <str name="f.price.facet.range.gap">50</str>
  <str name="q.alt">*:*</str>
  <str name="f.content.hl.fragsize">200</str>
  <str name="v.layout">layout</str>
  <str name="echoParams">all</str>
  <str name="fl">*,score</str>
  <str name="f.price.facet.range.end">600</str>
  <str name="hl.simple.post">&lt;/b>&lt;/str>
  <str name="f.name.hl.fragsize">0</str>
  <arr name="facet.field">
    <str>cat</str>
    <str>manu_exact</str>
    <str>content_type</str>
    <str>author_s</str>
  </arr>
  <str name="hl.encoder">html</str>
  <str name="v.template">browse</str>
  <str name="spellcheck.alternativeTermCount">2</str>
  <str name="f.popularity.facet.range.end">10</str>
  <str name="f.manufacturedate_dt.facet.range.start">
```

```

NOW/YEAR-10YEARS</str>
<str name="spellcheck.extendedResults">false</str>
<str name="spellcheck.maxCollations">3</str>
<str name="hl.fl">content features title name</str>
<str name="f.content.hl.maxAlternateFieldLength">750</str>
<str name="spellcheck.collate">true</str>
<str name="wt">xml</str>
<str name="defType">edismax</str>
<str name="rows">10</str>
<str name="facet.range.other">after</str>
<str name="f.popularity.facet.range.start">0</str>
<str name="f.title.hl.alternateField">title</str>
<str name="facet.pivot">cat,inStock</str>
<str name="f.title.hl.fragsize">0</str>
<str name="spellcheck">on</str>
<str name="spellcheck.maxCollationTries">5</str>
<arr name="facet.range">
  <str>price</str>
  <str>popularity</str>
  <str>manufacturedate_dt</str>
</arr>
<str name="hl.simple.pre">&lt;b&gt;</str>
<str name="hl">on</str>
<str name="title">Solritas</str>
<str name="df">text</str>
<arr name="facet.query">
  <str>ipod</str>
  <str>GB</str>
</arr>
...
</lst>
...

```

此处为节省篇幅，省略了一些默认参数。

从代码清单 4.7 中可以看出，针对搜索请求处理器的请求参数修饰是 Solr 的一大功能亮点，特别是默认参数列表的设计，为应用程序带来两大好处：

- 在应用程序的某个配置文件中设置合理的默认参数值，可以帮助简化客户端的代码。例如，将响应读写器参数 wt 的值设为 velocity，这样客户端应用就不需要在请求语句中指定返回数据的格式了。更重要的是，如果需要将 Velocity 改为其他的模板引擎，客户端代码不需要对此做任何改动。
- 通过预先配置复杂的组件，如分面，可以确保所有查询行为的一致性，同时保证客户端代码的简洁。从代码清单 4.7 中可以看出，示例请求包含了 Solritas 中搜索组件的一些复杂参数配置。在 Solritas 中配置分面会涉及 20 多个参数。

/browse 请求处理器是展示 Solr 在查询处理方面的功能特性的很好载体，但是一般情况下，开发者的应用程序无法直接使用它，因为它的参数配置和 Solritas 的底层数据模型紧密联系在一起。例如，Solritas 中的区间分面字段是 price、

popularity、manufacturedate\_dt 等，这是针对 Solr 的数据模型设计的，可能并不适用于开发者的应用程序。因此，读者在为自己的应用设计请求处理器时，应该仅仅将 /browse 请求处理器看作一个样板，而不是一个可以 100% 复用的解决方案。

#### 4.2.4 利用搜索组件扩展查询处理

除了一组默认值以外，/browse 请求处理器还定义了一个数组 <arr>，其中包含用 <last-components> 定义的一组后处理组件。注意到在代码清单 4.6 中 /browse 请求处理器定义了

```
<arr name="last-components">
  <str>spellcheck</str>
</arr>
```

这样的配置意味着，执行完默认搜索组件之后，再执行拼写检查组件。这是搜索请求处理器的一种常见设计模式。图 4.7 展示了在查询处理的 <components> 阶段使用到的 6 个内置搜索组件。

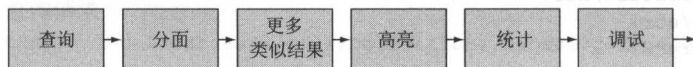


图 4.7 由 6 个内置搜索组件组成的处理链

##### 查询组件

查询组件是 Solr 查询处理过程的核心。从较高层次看，查询组件利用处于活跃状态的搜索器对查询语句进行解析与执行。搜索器的相关内容将在 4.3 节中讨论。查询语句的解析策略由 defType 参数指定。例如，/browse 请求处理器指定 eDisMax 作为查询语句的解析器（<str name="defType">edismax</str>），eDisMax 的相关内容将在第 7 章中详细介绍。

查询组件在索引中找出所有符合条件的文档，形成结果文档集。结果文档集可以随后供查询处理链中的其他组件（如分面组件）使用。查询组件默认处于启用状态，而所有的其他组件则需要查询请求中指定相应的参数来启用。

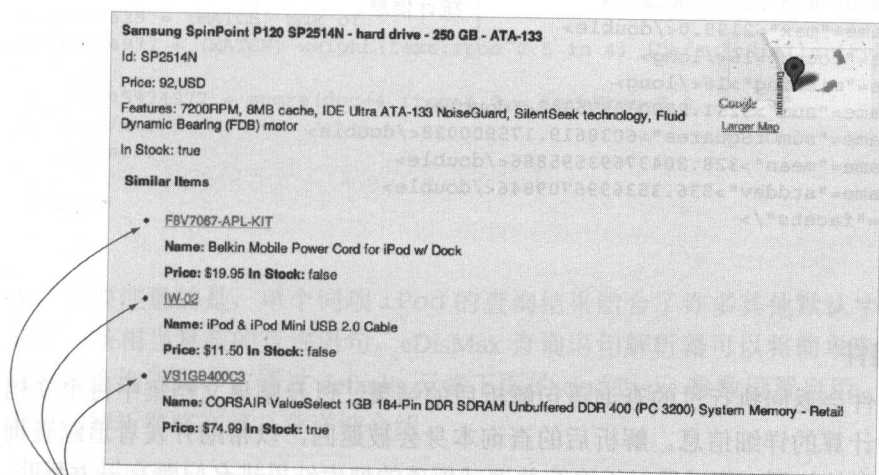
##### 分面组件

当给定一组由查询组件得出的结果文档集时，如果分面组件被启用了，它将根据字段分面进行结果统计与过滤。第 8 章将深入介绍分面的相关内容。目前需要了解的是分面已经内置在每个搜索请求中，但需要在查询请求中指定对应参数才能被

启用。在 /browse 请求处理器中, 分面是通过配置文件中的默认参数配置来启用的:  
<str name="facet">true</str>。

### 更多类似结果组件

当给定一组由查询组件生成的结果文档集时, 如果更多类似结果组件被启用了, 它将识别出与搜索结果集中的文档相似的其他文档。下面是一个更多类似结果组件的实例, 在 Solritas 示例应用中搜索 hard drive, 点击搜索结果 “Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133” 的 More Like This 链接, 可以看到一个相关文档的列表, 如图 4.8 所示。



更多类似结果组件  
可以找到相似结果。

图 4.8 利用更多类似结果组件发现的相关项示例

关于更多类似结果组件的更多内容请参见第 16 章。

### 高亮组件

如果高亮组件被启用了, 它将对结果文档中与查询语句高度相关的文档内容进行高亮表示。详细内容请参见第 9 章。

### 统计组件

统计组件可以为结果文档中的数值字段计算最小值、最大值、总和、平均值和标准差等简单的统计指标。执行代码清单 4.8 中的 GET 请求可以得到统计组件的一个运行示例。

## 代码清单 4.8 使用统计组件计算的 price 字段的相关统计数据

## 查询请求

```
http://localhost:8983/solr/collection1/select?
```

```
q=*:*&
wt=xml&
stats=true&
stats.field=price
```

将 price 字段设置  
为统计字段。

## 搜索结果

```
...
<lst name="price">
  <double name="min">0.0</double>
  <double name="max">2199.0</double>
  <long name="count">16</long>
  <long name="missing">16</long>
  <double name="sum">5251.270030975342</double>
  <double name="sumOfSquares">6038619.175900028</double>
  <double name="mean">328.20437693595886</double>
  <double name="stddev">536.3536996709846</double>
  <lst name="facets"/>
</lst>
...
```

返回的 price 字段  
统计信息。

## 调试组件

调试组件会返回执行过的查询语句解析后的结果，以及结果文档集中每个文档相关度分数计算的详细信息。解析后的查询本身会被返回，以帮助开发者追踪查询表达式中存在的问题。调试器对于追踪并调试烦琐的搜索结果排名问题有很大帮助。在浏览器中访问 <http://localhost:8983/solr/collection1/browse?q=iPod&wt=xml&debugQuery=true> 可以查看一个调试组件的运行示例。

需要注意的是，这个 URL 请求所执行的查询与上文在 Solritas 示例中通过表单提交的查询是一样的，只是在这里将响应读写器参数 wt 设为 xml（而不是 velocity），并且在 URL 中指定 debug=true，即启用调试组件。代码清单 4.9 展示了调试组件返回的 XML 数据片段。

## 代码清单 4.9 调试组件返回的 XML 数据片段

## 查询请求

```
http://localhost:8983/solr/collection1/browse?
```

```
q=iPod&
wt=xml&
debug=true
```

启用调试组件。

## 搜索结果

```

...
<lst name="debug">
  <str name="rawquerystring">iPod</str>
  <str name="querystring">iPod</str>
  <str name="parsedquery">(+DisjunctionMaxQuery((id:iPod^10.0 |
    author:ipod^2.0 | title:ipod^10.0 | text:ipod^0.5 | cat:iPod^1.4 |
    keywords:ipod^5.0 | manu:ipod^1.1 | description:ipod^5.0 |
    resourcename:ipod | name:ipod^1.2 | features:ipod | sku:ipod^1.5)))//
    no_coord</str>
  ...
  <lst name="explain">
    <str name="IW-02">
      0.13513829 = (MATCH) max of:
      0.045974977 = (MATCH) weight(text:ipod^0.5 in 4) [DefaultSimilarity], result
        of:
        0.045974977 = score(doc=4,freq=3.0 = termFreq=3.0
      ), ...</str>
    </lst>
  ...
</lst>
...

```

eDisMax 查询解析器生成的查询。

解释每个相关文档的分数计算。

需要注意的是，单个词项 iPod 的查询结果结合了许多其他默认字段，从而组成了一条相当复杂的查询语句。eDisMax 查询语句解析器可以将简单查询语句转化成复杂查询语句，它通过 defaults 元素下面的 defType 参数设置启用。eDisMax 查询语句解析器将在第 7 章详细介绍。

### 添加拼写检查作为后处理组件

在 6 个内置的搜索组件处理完搜索请求之后，/browse 搜索处理器将调用拼写检查组件，该组件被定义在 <last-component> 中。代码清单 4.10 展示了 solrconfig.xml 中对拼写检查组件的定义。

代码清单 4.10 定义拼写检查组件

```

拼写检查的
具体参数，
参见第 10
章。
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
  <str name="queryAnalyzerFieldType">textSpell</str>
  <lst name="spellchecker">
    <str name="name">default</str>
    <str name="field">name</str>
    <str name="classname">solr.DirectSolrSpellChecker</str>
    ...
  </lst>
</searchComponent>

```

定义一个名为“spellcheck”的 solr.SpellCheckComponent 类型的搜索组件。

请注意，拼写检查组件的名称与 /browse 请求处理器中 <last-components> 组件的名称要保持一致。因为要想完全理解代码清单 4.10 中的内容，

需要具备一些 Solr 中拼写检查处理的背景知识, 所以该配置元素在第 10 章中会继续讨论。本章要掌握的重点是了解如何利用 `<last-components>` 将一个搜索组件添加到搜索请求的处理流程中。

学习到这里, 你应该对 Solr 如何处理查询请求有了深入了解。在进入下一个配置主题之前, 你应该了解, 在 Solr 管理控制台的 Plugins/Stats 页面的 QUERYHANDLER 选项中可以查看所有处理活跃状态的 Solr 搜索处理器。图 4.9 展示了 `/browse` 搜索处理器的属性和相关统计数据, 或许你已经猜到了, 这又是一个 MBean。

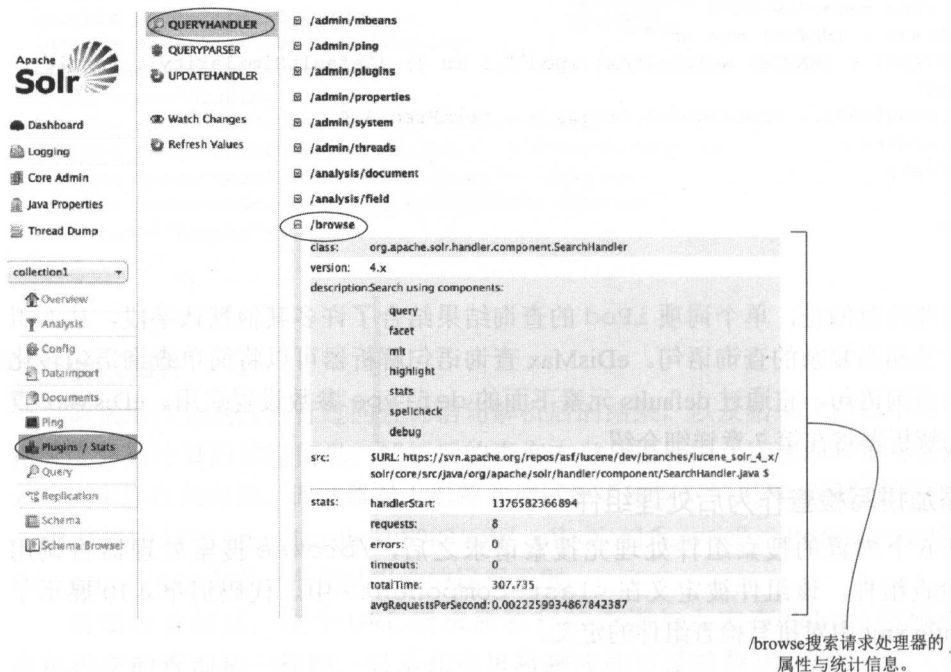


图 4.9 Solr 管理控制台的 Plugins/Stats 页面的 QUERYHANDLER 一栏中 `/browse` 请求处理器的属性与统计信息截图

下面将介绍一些优化系统性能的配置。

## 4.3 管理搜索器

`<query>` 中包含一些查询性能优化的配置, 这些配置主要利用缓存、字段延迟加载和新搜索器预热等技术。从一开始就考虑查询性能优化设计问题, 这对搜索应用的成功而言非常重要。本节介绍搜索器的管理技术, 这是查询性能优化中最重要的技术之一。



### 4.3.1 新建搜索器

Solr 的所有查询语句都由一个叫搜索器的组件处理。在 Solr 中，任何时候只能存在一个“处于活跃状态的”搜索器。所有搜索请求处理器中的查询组件都向这个处于活跃状态的搜索器发起查询请求。

处于活跃状态的搜索器拥有底层 Lucene 索引快照的只读视图。这意味着，如果现在将一份文档添加到 Solr 中，它在当前搜索器的搜索结果中是不可见的。这就引出一个问题：新添加的文档如何才能出现在搜索结果中？答案是关闭当前的搜索器，打开一个新的搜索器，这个搜索器添加了索引更新以后的只读视图。Solr 中的提交操作比这里描述的要复杂得多，相关的内容将在下一章进一步讨论。现阶段，将提交操作看成一个黑箱操作即可，该操作可以开启一个新的搜索器，向 Solr 中添加新文档，更新已有索引，让添加的内容在搜索结果中可见。

图 4.10 展示了示例服务器中 collection1 的处于活跃状态的搜索器 MBean，点击 Plugins/Stats 页中的 CORE 一栏进行查看。

点击CORE链接，找到活跃的搜索器MBean。

collection1内核中活跃的搜索器相关属性与统计信息。

在collection1内核的插件/统计页面访问MBeans。

注意warmupTime的取值。

```

class: org.apache.solr.search.SolrIndexSearcher
version: 1.0
description: index searcher
src: $URL: https://svn.apache.org/repos/asf/lucene/dev/branches/lucene_solr_4_x/solr/core/src/java/org/apache/solr/search/SolrIndexSearcher.java $

stats:
  searcherName: Searcher@f0fba68 main
  caching: true
  numDocs: 32
  maxDoc: 32
  deletedDocs: 0
  reader: StandardDirectoryReader(segments_1:3nrt_0(4,4):C32)
  readerDir: org.apache.lucene.store.NRTCachingDirectory.NRTCachingDirec
    org.apache.lucene.store.NIOFSDirectory@
    /Users/thelabduke/dev/SolrAction/solr-
    4.x/example/solr/collection1/data/index
    lockFactory=org.apache.lucene.store.NativeFSLockFactory@
    59d794d; maxCacheMB=48.0 maxMergeSizeMB=4.0)
  indexVersion: 3
  openedAt: 2014-02-01T16:17:46.635Z
  registeredAt: 2014-02-01T16:17:46.637Z
  warmupTime: 1
  
```

图 4.10 从管理控制台查看 collection1 中处于活跃状态的搜索器 MBean

在 CORE 页，注意 searcherName 属性的值（在图中为 Seacher@25082661 main）。通过向服务器重新发送所有示例文档来创建一个新的搜索器，这与 2.1.4 节中的操作一样：

```
cd $SOLR_INSTALL/example/exampledocs
java -jar post.jar *.xml
```

现在，刷新 COER 页，注意观察 searcherName 属性值，它已经变为了另一个搜索器实例。此处一个新的搜索器被创建出来，是因为 post.jar 命令在添加文档之后提交了一个 commit 提交请求。

提交操作会创建新的搜索器，以确保更新后的文档和索引可以被搜索器检索到。首先，旧的搜索器必须被销毁，但旧搜索器上可能存在大量查询请求，所以，Solr 要等到所有正在进行的请求都被旧搜索器处理完之后才会执行销毁操作。

同时，所有基于当前搜索器中索引视图的缓存对象都将失效。下一节将介绍更多关于 Solr 缓存管理的内容。现在我们来考虑某个特定查询结果集的缓存，因为该缓存中的部分文档可能已经被删除，同时可能添加了新的文档，所以很明显该结果集的缓存不再适用于新搜索器。

因为之前计算的数据（如上文中的缓存查询结果集）必定是无效的，需要重新计算，所以基于索引开启一个新搜索器实际上是一个非常耗费资源的操作，这可能会对用户体验直接造成影响。想象一下，一个用户正在分页浏览搜索结果，一个新搜索器在他点击第二页之后、第三页之前开启了。当用户请求下一页的时候，所有之前计算的过滤条件和缓存文档都失效了，需要重新计算。这时如果处理不当，用户会感到明显的延迟，特别是他们的查询请求比较复杂的情况下。

所幸，Solr 有许多工具可以帮助解决这个问题。最重要的是，Solr 支持预热理念，即在后台预热新搜索器，同时保证当前搜索器的活跃状态，直到新搜索器预热完毕。

### 4.3.2 新搜索器预热

Solr 秉承一种理念，即允许应用在短时间内提供过期数据，但不允许应用的查询性能有大幅下降。也就是说，在新搜索器预热完成，并准备好处理查询氢气迁，Solr 不会关闭当前活跃的搜索器。

新搜索器的预热过程好比田径比赛中短跑选手们的热身过程。在选手们开跑之前，他们需要进行热身准备，以确保在指令枪响起后可以保持良好的全速状态。同样地，Solr 也需要对新搜索器进行预热。

一般而言，Solr 有两种预热机制：一种是利用旧缓存自动预热新缓存，另一种是执行缓存预热查询。我们将在下一节中详细介绍自动预热缓存，并探讨 Solr 的缓存管理功能。

缓存预热查询就是向搜索器提交一段预先在 solrconfig.xml 文件中配置好的查询语句，目的是让新搜索器将需要缓存的查询结果载入它的缓存中。代码清单 4.11 展示了示例服务器中缓存预热查询的配置。

代码清单 4.11 定义一个执行预热查询的 newSearcher 事件的监听器

内部注释掉的代码；为你的环境配置应用专属查询。

```
<listener event="newSearcher" class="solr.QuerySenderListener">
  <arr name="queries">
    <!--
    <lst><str name="q">solr</str><str name="sort">price asc</str></lst>
    <lst><str name="q">rocks</str><str name="sort">weight asc</str></lst>
    -->
  </arr>
</listener>
```

定义一个监听器，处理 newSearcher 事件。

定义一个查询对象的命名列表，预热新搜索器。

代码清单 4.11 在配置文件中定义了一个查询语句列表 (<arr name="queries">), 一旦 Solr 中有 commit 等 newSearcher 事件发生时, 这些查询语句就会被执行。同时, 注意代码清单 4.11 中注释掉的部分, 这是有意注释的, 以提醒开发者执行预热查询会消耗一定系统资源, 开发者需要根据应用程序来配置自己的预热查询语句。正是因为配置与具体应用程序有关, 所以代码清单 4.11 仅是举例, 开发者需要根据需求自行配置预热查询。

### 选择预热查询语句

只有识别出能够提高查询性能的查询语句时, 搜索器的预热才能真正地发挥作用。一般而言, 预热查询语句应当包含应用程序中使用最频繁的查询请求参数 (q、fq、sort 等)。由于本章没有介绍 Solr 的查询语法, 所以预热查询内容将在第 7 章中继续讲解, 包括如何构造预热查询语句。现阶段, 在脑海中留下简单印象即可, 对 Solr 查询构造有了更深的理解之后再回到当前主题。

应当指出的是, 应用程序并不一定都要有预热查询语句。如果在提交操作之后查询性能出现明显下降, 才有必要考虑使用预热查询语句。

### 太多预热查询会降低性能

俗话说“少即是多”, 对于预热查询来说也是这样。每一条查询语句的执行都要花费一定的时间, 所以配置过多的预热查询将在开启新搜索器时导致较长的延迟。所以最佳做法是将预热查询数量尽可能降到最少, 只包含应用程序中最重要的查询语句。

你可能会有这样的疑问, 新搜索器花费较长时间预热会带来怎样的问题? 实际上, 应用程序中如果并发预热太多的搜索器会严重地消耗系统资源 (CPU 和内存), 严重影响搜索体验。

### 第一个搜索器

在 Solr 初始化过程中或重新加载完一个内核之后, 预热第一个搜索器。是否为第一个搜索器配置预热查询语句, 这完全由开发者自己决定。大多数的 Solr 开

发者使用完全相同的查询语句来预热新搜索器和第一个搜索器。在介绍 Solr 的缓存管理之前，我们简单介绍两个在 `solrconfig.xml` 文件中与搜索器相关的配置元素：`<useColdSearcher>` 和 `<maxWarmingSearchers>`。

#### 使用 XInclude 从其他数据源中提取 XML 元素

Solr 使用 XInclude 将其他文件中的 XML 元素提取出来放入 `solrconfig.xml` 中，无须为新搜索器和第一个搜索器重复设置预热查询列表。例如，将该列表保存为单独的文件，通过 `<xi:include href="warming-queries.xml" xmlns:xi="http://www.w3.org/2001/XInclude">` 的 XInclude 引用包含它。

#### useColdSearcher

`<useColdSearcher>` 适用于一个搜索请求已经被提交，而目前 Solr 中没有定义搜索器的情形。如果 `<useColdSearcher>` 的值为 `false`，那么 Solr 将会一直处于阻塞状态，直到正在预热的搜索器执行完所有的预热查询。示例 Solr 服务器中的默认配置为 `<useColdSearcher>false</useColdSearcher>`。

如果 `<useColdSearcher>` 设为 `true`，则 Solr 会立即使一个正在预热的搜索器进入活跃状态，而不管搜索器的预热程度如何。还以之前田径赛场为例，`false` 意味着指令官等到运动员充分热身之后才发出开始指令，不管这需要等多久；`true` 则意味着比赛马上就要开始，不管运动员是否已经充分热身了。

#### maxWarmingSearchers

可以想象，在新搜索器预热完成之前很可能接到，这就意味着在当前的新搜索器预热完成之前，另一个搜索器的预热又开始了。如果搜索器的预热时间很长，那么上述情形就会频繁发生。`<maxWarmingSearchers>` 元素允许开发者控制后台并发预热的搜索器的最大数目。一旦达到阈值，新的提交请求将会失败。这是一种很好的保障机制，因为后台并发预热的搜索器过多时会快速耗尽服务器的内存和 CPU 资源。Solr 中该元素的默认值为 2，这是一个不错的初始值：

```
<maxWarmingSearchers>2</maxWarmingSearchers>
```

当然，如果发现服务器经常达到阈值，那就需要重新思考预热机制的设计逻辑，看看新搜索器的预热时间是否过长。

读到这里，你应该对搜索器是什么，以及如何配置 Solr 来正确管理应用中的搜索器有一定的理解了。下面将介绍利用缓存对查询性能进行优化的方法。

## 4.4 缓存管理

Solr 提供了一系列的内置缓存来优化查询性能。在介绍某种缓存的具体细节之前，这里先介绍一下 Solr 的缓存原理。

### 4.4.1 缓存原理

Solr 的缓存原理主要涉及以下 4 个方面：

- 缓存大小及缓存置换法
- 缓存命中率与缓存回收
- 缓存对象失效
- 自动预热新缓存

一般来说，合理的缓存管理方式不是设置完以后便将其抛至一旁，而是要时时关注缓存的使用状况，基于实际使用情况进行调整。记住，要善于利用 Solr 的管理控制台来调试缓存、搜索器等重要组件。

#### 缓存大小

从缓存大小的角度来看，不能将缓存设置得太大，否则它会消耗 JVM 中的所有内存。Solr 能够将所有缓存的对象都保存在内存中，不会溢出到硬盘上，而这种情况在其他缓存管理框架中有可能会发生。为了控制缓存大小，Solr 要求为每一个缓存都设置一个缓存对象的数量上限。当达到上限时，Solr 将采用最久未使用（Least Recently Used, LRU）置换法或最近最少使用（Least Frequently Used, LFU）置换法回收一部分缓存空间。

最久未使用置换法在缓存大小达到阈值上限时，根据缓存对象最后一次被请求的时间决定缓存对象被回收的次序。当缓存区已达上限，需要添加新的对象时，LRU 置换法将置换缓存中最“老”的对象，即最久未被请求过的对象。LRU 置换法是 Solr 的默认缓存置换算法。

同时，Solr 还提供了最近最少使用置换法 LFU，该方法根据缓存对象被请求频率的高低决定缓存对象被回收的次序。这种置换法给予缓存中被调用频率高的对象更高的优先级，而不是最近被请求的对象。Solr 的过滤器缓存是使用 LFU 置换法的一个好例子，因为过滤查询的创建和存储是很耗费资源的，所以需要尽量降低过滤器缓存的存储大小，并且给予应用中频繁被调用的过滤器更高的优先级。下一节将介绍更多关于过滤器缓存的内容。

关于缓存大小的一个常见误区是，如果有比较大的内存空间，就应该将缓存空间也设置得比较大才行。这样做存在的问题是，一旦某个缓存在一次提交操作之后失效了，JVM 就需要做大量的垃圾回收工作。记住，关闭一个搜索器会使得该搜索器缓存的所有对象都失效。如果没有根据垃圾回收的实际情况对缓存大小进行合适的调整，就可能导致服务器因垃圾回收而长时间暂停服务。我们将在第 12 章中更详细地介绍 Solr 的垃圾回收参数。现阶段，最重要的是避免定义过大的缓存区，并且要让对缓存对象进行周期性回收。

### 缓存命中率与缓存回收

缓存命中率是指应用程序的缓存命中的用户请求数量占有所有用户请求数量的比例。缓存命中率表明了缓存对应用程序的性能优化所起到作用。理想状况下，开发者都希望应用的缓存命中率尽量接近 1（100%）。低缓存命中率表明缓存对 Solr 的性能优化没有起到作用。

缓存回收数表明有多少缓存对象根据上文介绍的缓存置换法被回收了。如果缓存回收量很大，则表明应用程序的缓存对象数量的最大值可能设置得太小。缓存回收数和缓存命中率是紧密相关的，高的缓存回收数往往导致一个较好的缓存命中率。第 12 章将详细介绍缓存命中率的优化。

### 缓存对象失效

在大多数的缓存管理场景中，开发者需要考虑如何使一个缓存对象失效，这样应用程序才不会返回过时的数据。在 Solr 中不再需要为这点担心，因为所有缓存中的对象都会链接到对应的搜索器实例，并且在搜索器关闭后立即失效。上文已经介绍过，搜索器只是 Lucene 索引快照的一个只读视图，因此，所有的缓存对象在搜索器关闭之前都是有效的。

### 自动预热新的缓存

我们在 4.3 节中讨论过，Solr 在每次提交请求之后都会创建一个新搜索器，并且直到新搜索器完成预热，才会关闭旧搜索器。Solr 利用即将被关闭的旧搜索器中的部分缓存内容构建新搜索器的缓存，这个过程被称为自动预热。请注意，自动预热新的缓存与使用预热查询预热新的缓存是两个不同的概念，这点我们已经在 4.3.2 小节中介绍过。

每一个 Solr 的缓存都支持 `autowarmCount` 属性，这个属性表示自动预热的旧缓存对象的最大数目或百分比。缓存对象自动预热策略取决于缓存的具体类型。本



章接下来介绍不同类型缓存的自动预热策略。现阶段的重点是学会 Solr 缓存的配置，当开启一个新搜索器时刷新部分缓存对象，至于上文中的优化技巧，需要注意不要过度运用它们。

学到这里，你应该对 Solr 中缓存管理的概念有了基本的理解。本章接下来将介绍 Solr 中实现查询性能优化的一些具体的缓存类型，先从最重要的一种缓存。

### 4.4.2 过滤器缓存

在 Solr 中，过滤器将搜索结果限制在符合过滤条件的文档集中，但是它并不影响文档的评分。2.2.1 节中的第一个查询示例使用了过滤器查询，利用 `fq=manu:Belkin` 过滤出制造商(`manu`)字段值为 `Belkin` 的文档，如代码清单 4.12 所示。

代码清单 4.12 使用针对 `manu` 字段的过滤查询 `fq` 的查询示例

```
http://localhost:8983/solr/collection1/select?
```

```
q=iPod&
fq=manu:Belkin&
sort=price asc&
fl=name,price,features,score&
df=text&
wt=xml&
start=0&rows=10
```

← manu 字段  
设置过滤查  
询 fq。

当 Solr 执行代码清单 4.12 中的查询语句时，它会计算并缓存一个合适的数据结构，找到索引中符合过滤条件的文档。在示例服务器中，有两篇文档符合该过滤条件。

假设有一个过滤条件相同 (`fq=manu:Belkin`) 但查询内容不同的查询请求，例如 `q=USB`，如果应用程序在处理该查询请求时可以利用代码清单 4.12 中的查询结果，那么查询效率会得到大幅提高，这正是 Solr 过滤器缓存设计的初衷。下面举例说明 Solr 的过滤器缓存。在浏览器中打开 `collection1` 的 Query 页面，提交查询请求，如图 4.11 所示。

接下来，打开 `collection1` 的 Plugins/Stats 页面，点击 CACHE 链接，图 4.12 展示了过滤器缓存的 MBean 的属性和统计信息。多次重复执行相同查询，会看到过滤器缓存的统计信息在变化。



过滤器查询  
(限制搜索结果集的  
manu字段取值为Belkin)。

Request-Handler (qt)

/select

— common —

q  
iPod

fq  
manu:Belkin

sort  
price asc

start, rows  
0 10

fl  
name,price,features,score

df  
text

Raw Query Parameters  
key1=val1&key2=val2

wt  
xml

☒ indent  
☐ debugQuery

☐ dismax  
☐ edismax  
☐ hl  
☐ facet  
☐ spatial  
☐ spellcheck

Execute Query

图 4.11 利用过滤器缓存执行以 fq 为过滤条件的查询语句实例

通过一个很小的索引很难体会到过滤器缓存的价值，但是假如你有一份百万数量级别文档的索引，这时就能体会到过滤器缓存带来的查询性能优化。实际上，使用过滤器进行查询优化是 Solr 的一大亮点，主要是因为过滤器在不同查询语句之间可以被复用。第 7 章将进一步介绍 Solr 的过滤器查询，而本章主要介绍 solrconfig.xml 文件中过滤器缓存的配置。代码清单 4.13 展示了过滤器缓存的默认配置。

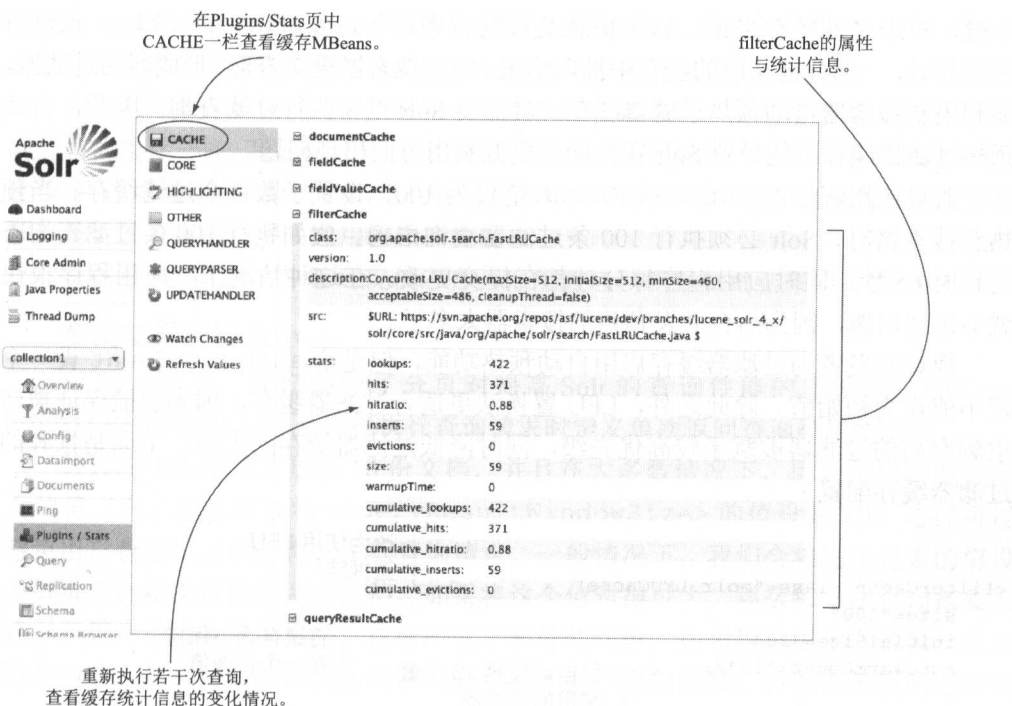


图 4.12 在 Plugins/Stats 页查看过滤器缓存的 MBean

## 代码清单 4.13 示例服务器中过滤器缓存的初始配置

```
<filterCache class="solr.FastLRUCache"
  size="512"
  initialSize="512"
  autowarmCount="0"/>
```

使用 LRY 策略。

该缓存没有对象进行  
预热。最大容量是 512  
个对象。

## 自动预热过滤器缓存

过滤器查询是优化查询语句的强大工具，但是如果开发者对过滤器缓存管理不当，则会陷入麻烦之中。如果索引中索引了大量的文档或者过滤条件非常复杂，那么创建和存储过滤器将耗费大量的系统资源。如果一个简单的过滤查询能够被复用于多条查询语句，那么这个过滤查询的缓存便是有意义的。除此之外，你可能还会想到在开启新搜索器时预热一些过滤缓存。

下面让我们深入到过滤器缓存的内部来看一看缓存对象的自动预热过程。读到这里，你应该理解了缓存对象并不能轻易地从旧缓存转移到新缓存中，因为底层的索引发生了改变，会使得过滤器之类的缓存对象失效。缓存中的每一个对象都有一

个键，对于过滤缓存来说，这个键就是过滤查询语句，如 manu:Belkin。预热新的缓存时，一部分键从旧的缓存中抽取出来，向新搜索器提交查询，形成新的过滤器。要利用新搜索器自动预热过滤器缓存，就需要 Solr 重新执行过滤查询。因此，自动预热过滤器缓存可能导致 Solr 在性能和资源利用方面出现问题。

假设应用程序的 autowarmCount 值设为 100，设置了数百个过滤缓存。当预热新搜索器时，Solr 必须执行 100 条过滤器查询语句。假如执行 100 条过滤查询语句耗时 65 秒，同时应用程序每分钟都在提交更改。在这种情况下，应用程序很快就会出现問題，因为后台并发预热的搜索器太多了。

建议开发者为过滤器缓存启用自动预热功能，但是给 autowarmCount 设一个较小值作为初始值。除此之外，LFU 置换法更适合过滤器缓存，因为它能保证被请求频率高的过滤器被赋予较高优先级，同时降低过滤器缓存的大小。下面是推荐的过滤器缓存配置：

```
<filterCache class="solr.LFUCache"
  size="100"
  initialSize="20"
  autowarmCount="10"/>
```

← 改为使用 LFU 置换法

← 将缓存大小保持在一个合理值

← 仅自动预热 10 个最常用的过滤器

开发者需要在此配置基础上，根据应用程序实际使用的过滤器数量和对索引提交更新的频率进行实验，以得到最佳配置。

从每个过滤器缓存的内存使用情况来看，根据匹配文档集合的大小，Solr 有不同的过滤表示方法。至于最大阈值，可以设定为每个过滤匹配文档集合中的 MaxDoc（最大文档数目）值。例如，如果索引文件索引了 1000 万份文档，那么一个过滤器最高可能占用一千万比特的内存，即 1.2MB 左右。

### 4.4.3 查询结果缓存

查询结果缓存会将查询请求的结果集保存在缓存中。如果多次执行代码清单 4.12 中的查询语句，实际上后面几次的查询结果都是第一次查询结果的缓存，而不是重新对 Lucene 索引执行查询。对于需要消耗大量计算资源的查询来说，这是一种非常高效的解决方案。查询结果缓存的定义如下：

```
<queryResultCache class="solr.LRUCache"
  size="512"
  initialSize="512"
  autowarmCount="0"/>
```

该方案背后的原理是将查询语句作为键，内部 Lucene 文档 ID 作为值，存储在

查询结果缓存中。内部 Lucene 文档 ID 会随着搜索器的改变而改变，所以在预热查询结果缓存时，缓存的内部 Lucene 文档 ID 需要重新计算。

为了预热查询结果缓存，Solr 需要重新执行查询语句，这可能消耗大量的资源。同过滤器缓存一样，我们建议将 `autowarmCount` 的值设为一个较小值。也就是说，建议将 `autowarmCount` 的值设为默认值 0 以外的某个较小值，这样可以通过自动预热最近的查询语句提高应用程序的性能。

除缓存大小之外，Solr 还提供各种其他配置选项调整查询结果缓存的使用。

### 查询结果窗口大小

在 2.2.4 节中，我们强调了分页对提高 Solr 的查询性能的重要性。`<queryResultWindowSize>` 允许在执行查询请求时定义单次返回查询结果的页数。

假设应用程序每页展示 10 份文档，并且在大多数情况下，用户都只浏览第一页和第二页。那么可以将 `<queryResultWindowSize>` 的值设为 20，这样可以避免用户查看第二页时再次执行查询请求。一般情况下，我们会将这个元素的值设为每页查询结果数量的两到三倍。如果将这个值设得过大，那么每一次查询都要加载多于用户请求数量的文档，这就产生了额外的代价。如果用户很少查看第一页以外的内容，那么最好将该元素的值设为单页查询结果的数量。开发者可以通过查看 Solr 日志文件中 `start` 参数的值是否大于 0 来决定该元素的值。

### 查询结果缓存的最大文档数

我们在 4.4.1 节中讨论过，必须为 Solr 的缓存大小设置最大值，但是这并不影响每个缓存对象在缓存中的大小。可以预见，一个包含一组百万级别文档结果集的缓存对象会肯定会影响 Solr 的可用内容。`<queryResultMaxDocsCached>` 元素允许对查询结果缓存中每个缓存对象包含的文档数目做出限制。在大多数搜索应用中，用户一般仅查看前几页的搜索结果，所以可以将这个值设为每页结果文档数目的两倍或三倍大小。

### 启用字段延迟加载

Solr 中一种常见的设计模式是只返回用户查询请求中要求的字段，而不是返回文档的所有字段。例如，代码清单 4.12 的示例查询语句请求了 `name`、`price`、`features` 和 `score` 字段，但是索引中的文档还有很多其他字段，如 `category`、`popularity`、`manufacturedate` 等。如果应用程序要采用这种设计模式，需要将 `<enableLazyFieldLoading>` 元素的值设为 `true`，这样可以避免加载用户不需要的字段。随书附带的示例代码索引的文档并没有太多的字段，所以这项设置带来的性能优化效果很容易被忽略。但在实际应用中，只返回用户需要的字段是一种高效的设计模式。

### 4.4.4 文档缓存

因为查询结果缓存仅仅缓存了一组符合查询条件的文档的内部 ID，所以即使查询结果被缓存了，Solr 仍然需要从硬盘中加载文档内容。文档缓存以文档的内部 ID 为键，将硬盘中的文档内容加载到缓存中。这样，查询结果缓存可以从文档缓存中调用需要的文档内容。

这就出现一个问题，即预热文档缓存是否有必要。一个很好的反面理由是，无法确定被预热的文档与来自查询结果和过滤器缓存的自动预热查询和过滤器存在什么关系。如果更新索引的频率很高，每个查询返回的结果文档也在经常变化，则配置文档缓存就可能把资源耗费在了对应用程序性能无益的地方。但另一方面，如果索引更新频率很低，那么文档缓存可能有助于提高应用程序的性能。

### 4.4.5 字段值缓存

最后要介绍的缓存是字段值缓存，主要受 Lucene 控制，而不是由 Solr 来管理。字段值缓存提供了通过内部文档 ID 快速访问存储的字段值的途径，主要在排序和从匹配文档中生成响应内容时使用。这是一个相对高级的话题，推荐参阅 Lucene JavaDoc (`org.apache.lucene.search.FieldCache`) 来获取更多信息。

学到这里，你应该了解了 Solr 处理客户端查询请求的过程，并且了解了如何利用预热新搜索器和缓存来提高应用程序的查询性能。

## 4.5 其他配置选项

截至目前，本章已经介绍了 `solrconfig.xml` 文件中最常见的配置，但是 Solr 还有很多其他特性。Solr 拥有许多专家级的配置，这些配置内容并不在本书的介绍范围之内。Solr 的示例配置文件 `solrconfig.xml` 对开发者来说很友好，因为文件中详细介绍了每项配置所控制的内容和配置方法。建议你多花一些时间好好研究该文件，学习更多的提高 Solr 性能优化方法和更多未启用的新特性。

因为 `solrconfig.xml` 是启用新特性必不可少的文件，所以本书后面还会讲到它。在本书的余下部分，你会看到如何在 `solrconfig.xml` 文件中启用 Solr 的新特性，如何设置响应数据的格式，如何定义新搜索处理器，如何自定义搜索组件，如何配置 Solr 的生产环境，以及其他与性能优化相关的内容。总之，Solr 的配置还远远没有介绍完，但这里你只需知道在初次配置和运行 Solr 时，哪些配置选项是可用的，以及哪些选项是重要的。

## 4.6 本章小结

通过学习本章,你应该对 Solr 的配置有了深刻的理解,尤其是 Solr 应用程序的查询处理优化。具体来说,你应该了解了 Solr 的查询处理管道,该管道由一个统一请求分配器和一个可灵活配置的处理请求器组成。搜索处理器分为四个组件,每个组件都可以由开发者自定义。Solr 应用程序中的 `/browse` 请求处理器是一个使用默认配置参数和自定义组件(如 `spellcheck`)的典型示例,该示例在为丰富搜索体验的同时,还保证了客户端代码的简洁。

另外,本章还介绍了 Solr 应用程序利用搜索器处理查询请求的原理,其中,搜索器基于 Lucene 索引视图。任时刻, Solr 中都只能存在一个处于活跃状态的搜索器,并且对 Lucene 索引的任何更新都需要在开启新的搜索器之后才能得到应用。但无论是关闭当前的搜索器,还是开启新的搜索器都要消耗大量的系统资源,影响到 Solr 应用程序的查询性能。为了使开启新的搜索器对查询性能的影响降到最低, Solr 允许配置静态的查询语句,利用这些查询语句在后台预热新的搜索器。对新搜索器的预热过程进行合理管理是 Solr 应用程序中最重要的配置任务之一。Solr 的其他生产环境中的配置将在第 12 章中介绍。

另外, Solr 提供了一系列重要的缓存,这些缓存需要根据应用程序的实际需求做适当的配置。本章重点介绍了过滤缓存、查询结果缓存、文档缓存和字段值缓存。对于每一种缓存,都需要根据应用程序的实际运行状况设定缓存大小的阈值,以及缓存对象的置换法。通过 Solr 内置的管理控制台,可以查看一些应用程序的关键统计信息,如缓存命中率等,这些统计信息可以帮助开发者设置合理的缓存大小。

当创建新搜索器时,为了优化应用程序的查询性能,可以预热缓存。例如,可以对被请求频率最高的过滤查询使用 Solr 中过滤器缓存的自动预热功能。尽管缓存的自动预热功能很强大,但是也会导致新搜索器预热的延时。建议将 `autowarmCount` 的初始值设为一个较小值,并且密切监视搜索器的预热时间。第 12 章将进一步介绍 Solr 中缓存的调整。

尽管本章中介绍的 Solr 配置选项基本覆盖了一般情况下的需求,但是本章并没有将 Solr 配置的所有内容都覆盖到(不过这也不是本章或本书的目标)。本章的目的只是想让读者熟悉 Solr 的配置,以及这些配置背后的原理,至于大量额外的配置选项及性能优化方案,将在本书的余下章节结合 Solr 的相应特性再做介绍。

正如本章一开始时所声明的那样, `solrconfig.xml` 文件中索引相关的配置将在读者理解了 Solr 的索引原理后再做介绍,详见下一章。

# 5 创建索引

## 本章要点

- 为待索引的文档设计模式
- 在 `schema.xml` 中定义字段和字段类型
- 对结构化数据使用字段类型
- 处理更新请求、提交和原子更新
- 在 `solrconfig.xml` 中管理索引配置

第3章介绍了 Solr 如何使用倒排索引查找文档。倒排索引最简单的形式是由词项字典和出现每个词项的文档列表两部分构成。Solr 使用倒排索引将用户查询中的词项与文档中出现的词项进行匹配。本章介绍 Solr 如何对文档创建索引，而创建索引中一个重要工作是文本分析。本章具体讲解索引的创建过程和非文本字段，第6章将详细讨论文本分析。

学完本章，你将了解如何在 Solr 中对文档进行索引，掌握字段、字段类型和模式设计等核心概念。如果你已经将第2章介绍的 Solr 示例服务器运行在本地计算机上，那么这一章内容掌握起来会容易一些。当然，如果没有运行 Solr 的话，先阅读本章，然后再动手操作，你也能搞清楚大多数示例。



## 5.1 微博搜索应用示例

贯穿本章，我们将通过索引创建与文本分析，设计并实现一个针对流行社交媒体网站（如 Twitter）的微博搜索。你可以把这里的微博当作为一个代名词，指代那些较短的、非正式消息及人们在社交网络上相互分享的内容。微博示例来自 Twitter 的推文、Facebook 的帖子、Foursquare 的签到信息等。本章将介绍如何在 Solr 中通过定义字段和字段类型来表示微博，以及如何将文档添加到 Solr。

第 6 章将介绍如何使用 Solr 内置工具对微博内容进行文本分析。首先，让我们看看这些要处理的文档和用户可能想要搜索它们的方式。

### 5.1.1 面向搜索的内容表示

首先，表 5.1 列举了一条虚构的微博应有的一些字段，接下来我们会使用这些内容来学习在 Solr 中如何索引文档。即便你对社交媒体内容分析不感兴趣，那也没关系，这个示例的学习经验适用于大多数搜索应用。

表 5.1 虚构微博的一些字段

字段	取值
id	1
screen_name	@thelabdude
type	post
timestamp	2012-05-22T09:30:22Z
lang	en
user_id	99991234567890
favorites_count	10
text	#Yummm :) Drinking a latte at Caffé Grecco in SF's historic North Beach. Learning text analysis with #SolrInAction by @ManningBooks on my iPad

Solr 索引中的每个文档由字段组成，每个字段拥有具体类型，字段类型决定了文档将如何被存储、搜索与分析。在表 5.1 中，微博文档有 8 个字段。<sup>1</sup> 思考一下，用户使用这些字段寻找微博的可能方式。从搜索角度看，我们认为 screen\_name、type、timestamp、lang 和 text 这些字段是不错的索引字段备选，它们包含了典型用户在构造查询时用到的内容。例如，一个用户可能想要查看特定

<sup>1</sup> 有关微博可用字段的详细讨论，建议阅读 Map of a tweet (<http://www.slaw.ca/wp-content/uploads/2011/11/map-of-a-tweet-copy.pdf>)。

用户 (screen\_name:tehlabdude) 在特定时间 (timestamp:[2012-05-01T00:00:00Z TO \*]) 以后的所有英文微博 (lang:en)。

当然,也可以索引所有的字段。但是如果开发一个支持百万级别文档与高查询量的大规模系统,那么只用包含用户会搜索的那些字段即可。例如, user\_id 字段是 Twitter 的内部识别符,用户不大可能想要搜索这个字段。一般情况下,每增加一个字段就会增加索引的体积,因此在索引中仅包含用户会搜索的字段即可。

favorites\_count 字段是微博作者的粉丝数,不是该条博主的点赞数。这个字段很有趣,从 UI 角度看,它提供了有用的信息,但不适合作为搜索查询参数的备选。在 5.2 节介绍存储字段与索引字段时,会顺便讨论这些以显示为目标的字段处理方法。

让我们想想用户如何使用这些字段来构造查询,这有助于确定如何在 Solr 索引中表示这些字段。图 5.1 展示了一个虚构的搜索表单,其中的字段来自之前讨论的微博搜索应用。

Microblog Search Tool

http://example.com/SolrInAction/TextAnalysis

Social Media Search Form

By User: @theIabdude

Type: Post, Reply, Retweet

Language: English

Date Range: After 05/01/2012

With Text: San Francisco north beach coffee

Search

Search Results

@theIabdude on May 22, 2012 @ 09:30 AM

#Yummm :) Drinking a latte at Caffé Grecco in SF's historic North Beach... Learning text analysis with #SolrInAction by @ManningBooks on my iPad

Favorited by 10 users

用户通过搜索表单可以搜索到之前索引过的所有微博文档字段。

用户名

类型

语种

时间戳

微博全文

当用户进行搜索时,根据文本分析,以下是微博搜索的一条示例结果。

图 5.1 虚构的微博搜索表单,用到了 screen\_name、type、lang、timestamp 和 textfields 字段

从搜索角度看,我们认为搜索表单里的每个字段都是有用的。搜索应用设计的一个关键是,考虑用户如何对索引中的特定字段进行搜索。这一点很关键,这也有助于确定在 Solr 中定义哪些字段。

现在，我们已经从概念上理解了微博搜索示例中的字段，了解到用户如何使用这些字段搜索文档。接下来，我们从宏观层面理解如何向 Solr 添加文档。

### 5.1.2 Solr 索引构建概览

从宏观层面看，Solr 的索引构建可分解为三个主要任务：

1. 将文档从原始格式转换为 Solr 支持的格式，例如，XML 或 JSON。
2. 从良好定义的接口方法中选择一种，通常使用 HTTP POST，将文档添加到 Solr。
3. 在索引中，通过配置 Solr，对文档的文本进行转换。

图 5.2 展示了 Solr 索引文档过程的宏观概览，包括三个基本步骤。

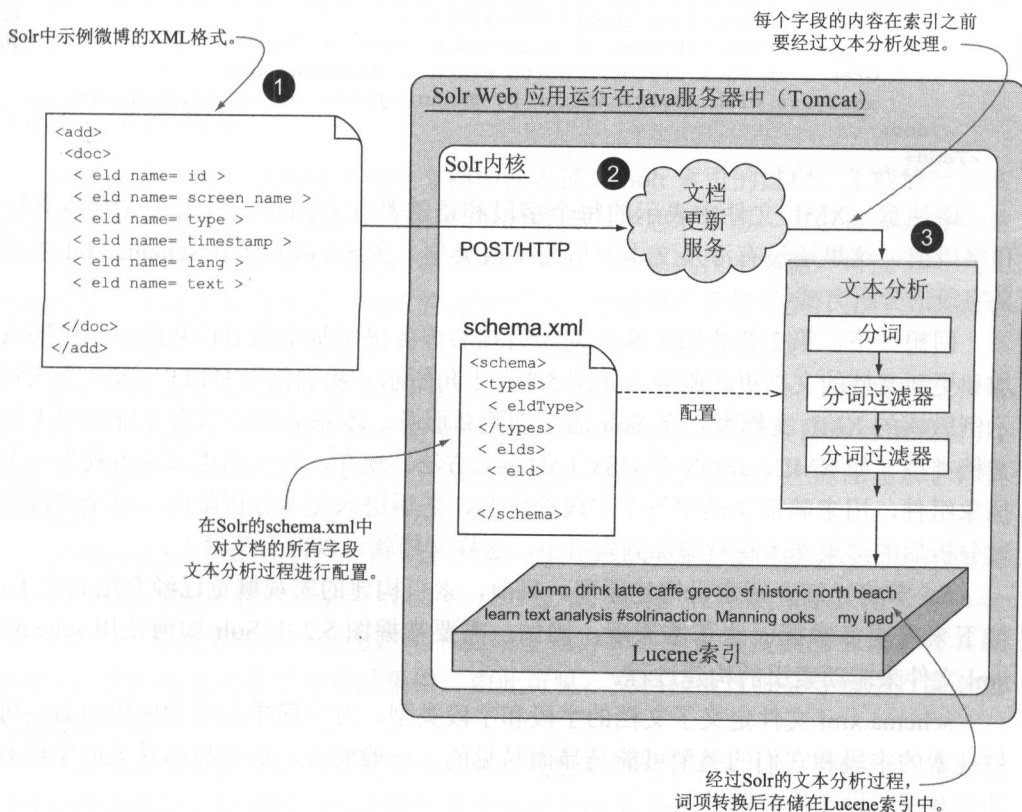


图 5.2 索引文档的三步概览。第一步，将微博表示为 Solr 支持的 XML 格式；第二步，使用 HTTP POST 方式将 XML 文档发送给 Solr 的文档更新服务；第三步，根据 schema.xml 文件中定义的配置，在添加到索引之前分析每个字段

Solr 的索引构建支持多种文档格式, 包括 XML、JSON 和 CSV。由于 XML 的自描述格式易于理解, 所以在图 5.2 中使用 XML 表示。示例微博在 Solr 中的 XML 格式详见代码清单 5.1。

代码清单 5.1 用于索引示例微博的 XML 文档

```

<add>
  <doc>
    <field name="id">1</field>
    <field name="screen_name">@thelabdude</field>
    <field name="type">post</field>
    <field name="timestamp">2012-05-22T09:30:22Z</field>
    <field name="lang">en</field>
    <field name="user_id">99991234567890</field>
    <field name="favorites_count">10</field>
    <field name="text">#Yummm :) Drinking a latte at Caffé
      Grecco in SF's historic North Beach... Learning text
      analysis with #SolrInAction by @ManningBooks on my i-Pad</field>
  </doc>
</add>

```

每次可以添加多个文档, 每个文档封装在 <doc> 元素中。

告诉 Solr 我们正在向索引中添加一个新文档。

提供文档中每个字段的名称和取值。

请注意, XML 文件中表示的每个字段和语法都非常简单, 只需定义字段名称和字段值。这里还没有涉及文本分析与字段类型。图 5.2 展示了在 schema.xml 中定义字段的分析方式。

回想一下, 第 2 章介绍了 Solr 对所有核心服务提供基本的 HTTP 接口, 包括添加和更新文档的文档更新服务。如图 5.2 左上角所示, 我们使用 HTTP POST 方式将示例微博的 XML 文档发送至 Solr 的文档更新服务。本章后续小节会详细介绍不同文档类型 (如 XML、JSON 和 CSV) 的添加方法。现在, 将文档更新服务视为一个抽象组件, 用来验证文档中每个字段的内容, 之后进入文本分析阶段。每个字段经过分析后的结果文本将被添加到索引中, 这样文档就可以用于搜索了。

5.5 节将详细介绍索引构建原理。目前, 索引构建的宏观概览已经介绍到位了, 接下来需要多掌握一些基本原理, 比如, 需要掌握图 5.2 中 Solr 如何使用 schema.xml 文件来驱动索引的构建过程。

schema.xml 文件定义了文档的字段和字段类型。对于简单的搜索应用而言, 可供搜索的字段和它们的类型可能是显而易见的。一般而言, 这些显而易见的字段有助于对模式设计的前期规划。

## 5.2 设计自己的schema

在微博搜索应用示例中，我们定义了文档的内容和希望索引的字段。但在实际操作层面，对于一个真实搜索应用而言，这个过程不总是显而易见的，所以做一些前期设计与工作规划是很有帮助的。本节介绍搜索应用应考虑的关键设计问题。具体而言，我们应该对搜索应用的以下关键问题做出应答：

- 索引中的文档内容有哪些？
- 每个文档如何做到唯一识别？
- 用户会搜索文档中的哪些字段？
- 在搜索结果中应该向用户显示哪些字段？

针对特定搜索应用，我们先从如何决定合适的文档粒度入手，因为这个问题会影响到其他问题。

### 5.2.1 文档粒度

schema 的设计过程实际是确定文档如何表征为 Solr 索引的过程。某些情况下这个过程是显而易见的，例如，我们示例中的微博内容通常很短，所以每条推文就是一个文档。然而，如果待索引的内容量大，例如，计算机技术书籍，可能要将大文档的各个子节视为索引单元。问题的关键在于，考虑用户希望看到什么样的搜索结果。让我们来看一个不同的例子，以便更好理解索引中的文档构成。

假设在出售计算机技术书籍的网站上搜索 "text analysis"。如果网站将每本书视为一个文档，用户会在搜索结果中看到《Solr 实战》，但需要翻阅目录或索引，找到书中出现 "text analysis" 的具体位置。如图 5.3 所示，左侧图显示了将整本书视为一个文档进行索引后的搜索结果。

如果网站将每本书的章节作为索引的文档，那么搜索结果可能会在一开始向用户显示《Solr 实战》的 "Text analysis" 章节，如图 5.3 右侧所示。由于 "text analysis" 是搜索的核心概念，本书的很多地方和其他有关搜索的书籍都与其高度相关，因此都出现在了搜索结果中。由此可见，文档粒度过细会导致用户面对过多的搜索结果。

内容类型也是索引创建时需要考虑的。按照章节对计算机技术书籍的内容进行拆分是行得通的，但对科幻小说进行章节拆分，似乎并不合适。也就是说，索引中的文档可以由你自主决定，但要充分考虑文档粒度对用户搜索体验造成的影响。通常，文档粒度的选择尽量不要让用户感到“见木不见林”。

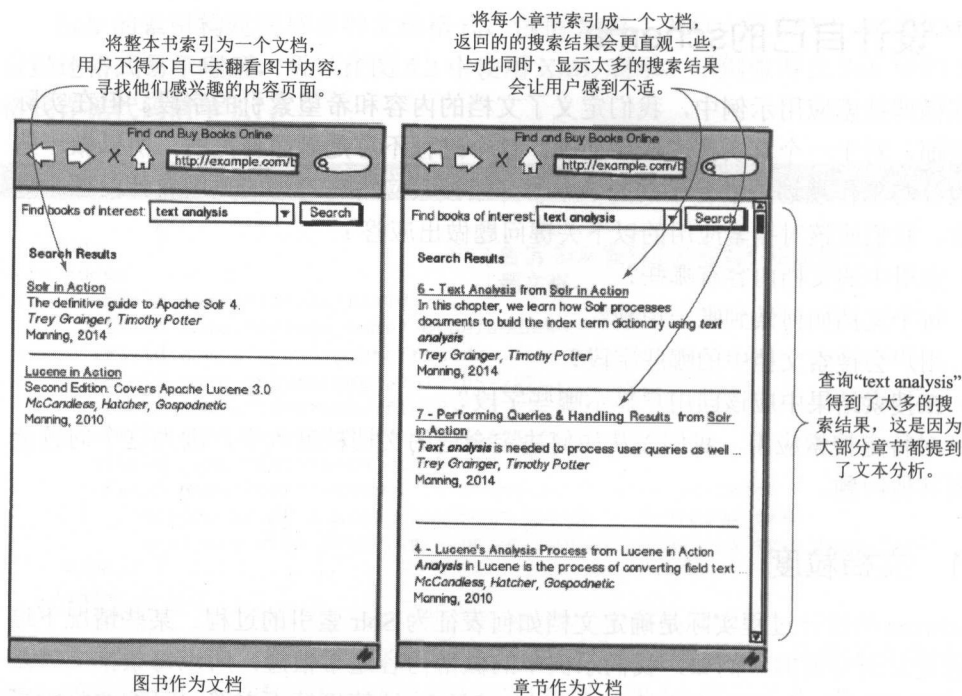


图 5.3 索引整本书和索引章节的搜索结果比较

### Solr 搜索结果高亮

另外需要留意，Solr 提供搜索结果高亮功能，对搜索结果中长文档的相关部分进行高亮显示。当无法将长文档分成小单元，但仍希望帮助用户在长文档中快速定位高度相关的章节时，高亮功能非常有用。例如，使用高亮对包含 "text analysis" 的文档显示该短语周围的文本短片断。第 9 章将讨论搜索结果高亮。

## 5.2.2 唯一键

一旦确定了待索引的文档构成，接下来就需要确定如何在索引中唯一识别每个文档。对于整本书，可能是 ISBN 号；对于章节，可能是 ISBN 号加上章节号。Solr 不要求每个文档都有一个唯一识别符，但若提供，Solr 会将其用于去除索引中的重复文档。如果你了解数据库，应该知道唯一识别类似于二维表中元组的主键。如果将相同主键的文档添加到索引，Solr 会用最新的文档覆盖已有的文档。5.3.5 节将继续讨论唯一键。

5.1 节微博搜索应用中的微博已经包含一个唯一识别符字段：id。如果索引的内容来自不同社交媒体，可能会出现相同 ID 的文档，那么需要增加前缀进行区分，例如，加前缀 twitter：以示与 Facebook 区分。



### 5.2.3 索引字段

确定了索引的文档构成以及文档唯一识别符后，接下来是确定文档中的索引字段 (*indexed fields*)。索引字段选择的最佳方法是，询问典型用户是否能使用该字段构造有意义的查询表达式。另一种方法是，如果搜索表单没有提供该字段的查询选项，但用户会提及它，那么这个字段就应该被索引。

例如，每本书都有书名和作者。搜索图书时，人们通常希望通过书名和作者找到感兴趣的书籍，因此这两个字段应该被索引。虽然每本书都有一个责任编辑，但读者找书时通常不会搜索编辑的名字，因此，编辑的名字无须作为索引字段。相反地，如果为图书出版业构建一个索引以供搜索，那么用户会搜索编辑的名字，这时可以将其作为索引字段。

为了启用搜索，需要对索引字段进行标记，还需要对字段值进行排序、分面、分组、查询建议及执行函数查询等。当启动了特定高级设置时（第9章会介绍），标记索引字段会有效提升搜索结果高亮处理效率。以上这些功能将在本书后续介绍，很多都以独立章节出现。从本质上讲，如果搜索结果中不止返回原始的字段值，还需对字段值进行运算的话，那么大多数情况下都需要索引这个字段。

对不同搜索应用而言，需要索引的字段都是不一样的。花点时间考虑一下文档的待索引字段，记录下来，随后会要用到这些字段。之前讨论过，`screen_name`、`type`、`timestamp`、`lang` 和 `text` 字段是微博搜索应该索引的字段。`id` 和 `user_id` 是 Twitter 的内部使用字段，如果不允许用户搜索它们，那就可以不理睬了。

### 5.2.4 存储字段

虽然用户可能不会通过编辑的名字去找要读的书，但我们仍希望在搜索结果中显示编辑的名字。一般情况下，文档可能包含了一些对搜索本身无用的字段，但这些字段显示在搜索结果中仍然是有用的。Solr 将这些字段称为存储字段 (*stored fields*)。`favorites_count` 字段是一个典型的存储字段，它不会被索引，只用于显示。设想一下，当用户在搜索结果中查看哪位作者更受欢迎时，`favorites_count` 就会很有用，但用户并不希望通过这个字段进行搜索。当然，一个字段可以同时被索引和存储，例如，微博搜索应用的 `screen_name`、`timestamp` 及 `text` 字段。每个字段都能被搜索和显示在搜索结果中。

作为一名搜索应用架构师，你的任务目标之一是尽量降低索引的大小。如果你正在考虑使用 Solr，那么你手头很可能有一个待扩容的应用，以应对大规模的文档和用户请求。索引中每一个存储字段都需要占用磁盘空间，请求 CPU 和 I/O 资源来读取存储值，将其返回在搜索结果中。应谨慎地选择存储字段，尤其是对于大规模应用而言。



基于这一点,设计搜索应用时需要认真思考所需的字段类型。一旦选定了一个方案,接下来就该着手在 schema.xml 中进行设计。如图 5.2 所示, schema.xml 是 Solr 的主要配置文件,从中可以了解如何进行文档索引。让我们预览一下 schema.xml 的主要部分,以便熟悉后续两小节中存储的字段情况。

### 5.2.5 schema.xml 概览

在接下来的几个小节中,我们为微博搜索应用设计一个有效的 schema.xml 文件。schema.xml 文件位于 Solr 内核的 conf/ 目录下。示例 Solr 服务器的 schema.xml 文件位于 \$SOLR\_INSTALL/example/solr/collection1/conf/ 中。代码清单 5.2 是 Solr 示例 schema.xml 的简要版,展示了 XML 语法和重要元素<sup>2</sup>。

代码清单 5.2 Solr schema.xml 文件的主要内容

版本号  
仅用于  
Solr 内  
部,启  
用具体  
功能。

```
<schema name="example"
  version="1.5">
  <fields>
    <field name="id" type="string" indexed="true" stored="true" .../>
    <field name="name" type="text_general" indexed="true" stored="true"/>
    <field name="cat" type="string" indexed="true" stored="true" .../>
    ...
    <dynamicField name="*_s" type="string" indexed="true" stored="true"/>
    <dynamicField name="*_t" type="text_general" indexed="true" .../>
    ...
  </fields>
  <uniqueKey>id</uniqueKey>
  <copyField source="cat" dest="text"/>
  ...
  <copyField source="manu" dest="text"/>
</types>
<fieldType name="string" class="solr.StrField" .../>
<fieldType name="boolean" class="solr.BoolField" .../>
...
<fieldType name="tint" class="solr.TrieIntField" .../>
<fieldType name="tfloat" class="solr.TrieFloatField" .../>
<fieldType name="text_general" class="solr.TextField">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" .../>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
```

← schema 名称仅用于显示目的。

← 字段元素表示文档中的字段。

← 定义哪个字段用于每个文档的唯一识别。

← 从一个字段向另一个字段复制字段值。

← fieldType 元素定义,以及如何分析字段。

← fieldType 配置有助于分析一般文本。

<sup>2</sup> 要查看完整内容,请在 Solr 管理控制台中点击内核左下角 schema 链接查看。

```

        <tokenizer class="solr.StandardTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" .../>
        <filter class="solr.SynonymFilterFactory" .../>
        <filter class="solr.LowerCaseFilterFactory"/>
    </analyzer>
</fieldType>
...
</types>
</schema>

```

初看之下，很容易被内容细节吓到。等读完本章，你就会对所有细节了如指掌，并能够创建自己的 schema.xml。schema.xml 文档由三个主要部分组成：

1. <fields> 元素包含 <field> 和 <dynamicField>，用来定义文档的基本结构。
2. 其他元素，如 <uniqueKey> 和 <copyField>，位于 <fields> 元素之后。
3. <types> 元素下的字段类型包括 Solr 能够处理的日期、数字和文本字段。

接下来依次介绍每个部分，首先是 <fields>。

## 5.3 在schema.xml中定义字段

schema.xml 的 <fields> 一节定义文档中所有用到的 <field> 元素。Solr 根据 schema.xml 中的字段定义来调用适合的字段分析器，将字段内容解析为词项，继而添加到倒排索引中(详见第 3 章内)。本节中我们学习如何在 schema.xml 中定义字段、动态字段与复制字段。5.2 节已经介绍了 schema 设计的主要理念，现在可以为微博搜索应用定义 <field> 元素了。代码清单 5.3 定义了示例搜索应用的索引字段和存储字段。

代码清单 5.3 微博搜索示例应用的字段定义

```

<schema name="example" version="1.5">
  <fields>
    <field name="id" type="string" indexed="true" stored="true"
      required="true"/>
    <field name="screen_name" type="string" indexed="true" stored="true"/>
    <field name="type" type="string" indexed="true" stored="true"/>
    <field name="timestamp" type="tdate" indexed="true" stored="true"/>
    <field name="lang" type="string" indexed="true" stored="true"/>
    <field name="favorites_count" type="int"
      indexed="true" stored="true"/>
    <field name="text" type="text_microblog"
      indexed="true" stored="true"/>
    ...
  </fields>
  ...
</schema>

```

该字段是索引中文档的唯一标识符。

该字段被索引是为了排序，被存储是为了显示。

在 <fields> 元素下定义微博搜索应用示例的字段。

该文本字段使用 text\_microblog 字段类型进行分析。

不论该字段是否被索引或存储，都必须定义名称和类型。

有了这些字段定义，Solr 就知道如何索引微博文档了，之后这些字段可以通过图 5.1 所示的表单进行搜索。在 schema.xml 中定义一个字段时还有一些必填属性。

### 5.3.1 必备字段属性

每个字段都有唯一名称，查询构造时会调用这个名称。例如，查询 screen\_name:thelabdude 搜索 screen\_name 字段，寻找字段值为 thelabdude。在代码清单 5.3 中，screen\_name 字段定义如下：

```
<field name="screen_name"
      type="string"
      indexed="true"
      stored="true" />
```

通俗点讲，此定义表示 screen\_name 字段既被索引又被存储，其包含字符串值。每个字段必须定义类型属性，通过 <fieldType> 识别字段类型。下一节将详细介绍字段类型。不论字段是否被索引或存储，每个字段都需要进行定义。正如 5.2 节讨论过的，索引字段可以被搜索和排序（其他功能将在本书其他地方介绍）。出于显示目的，存储字段也会返回在搜索结果中。一个字段可以同时被索引和存储，就像微博搜索应用示例的大多数字段情况。

另外，Solr 不允许嵌套字段。schema.xml 的所有字段都是同级的，也就是一个平面文档结构。正如第 3 章讨论过的，Solr 的文档需要进行非规范化处理，转化成平面结构，必须包含搜索需求所要求的所有字段。这意味着不存在关系结构，查询处理与搜索结果生成中无法连接其他文档以提供更多信息。

#### Solr 的连接操作

毫无疑问，你一定会在 Solr 中遇到文档连接情况，第 15 章会详细介绍连接功能。现在要清楚一点，Solr 的连接更像是 SQL 的子查询。一个典型的用例是找到符合搜索条件的文档的父文档。以微博搜索应用为例说明，就是通过 Solr 的连接找回转发微博的原始微博。

当一个字段被存储时可能会造成混淆，Solr 存储的是原始值，而非分析后的值。例如，代码清单 5.3 中将 text 字段定义为既可索引也可存储（indexed="true", stored="true"）。这意味着 text 字段可以被搜索，并在搜索结果中返回原始的文本。当请求字段返回值时，Solr 不会在搜索结果中返回分析后的值。当然，如果在搜索结果中不用返回这个字段，那么它就无须存储。

虽然随着索引大小的增长，查询速度会降低，但仍然有需要存储所有字段的情

况。如果字段已经被索引了，你又计划在 Solr 中对文档的字段进行更新，而不是从外部来源重新提交全部文档，那么就要存储所有的字段，让 Solr 保留每个字段原始内容的副本。5.6.3 节将会介绍如何更新文档。

5.3.2 多值字段

截至目前，微博搜索应用仅使用了几个简单字段。接下来，让我们添加和混合一些字段，来看看 Solr 在处理更复杂文档结构上的优势。具体来说，添加一个 links 字段，包含 0 个或多个与每个文档相关的链接。就像 Twitter 上那样，用户能插入相关内容链接，如网络上的一张照片或一篇文章。下面是虚构的嵌入两个链接的微博举例：

Just downloaded the ebook of #SolrInAction from @ManningBooks <http://bit.ly/T3eGYG> to learn more about #Solr <http://bit.ly/3ynriE>

以上微博中包含的短网址由 <http://bitly.com> 提供网页的解析跳转，如表 5.2 所示。

表 5.2 微博例子中短网址对应的真实网址

短网址	真实网址
<a href="http://bit.ly/T3eGYG">http://bit.ly/T3eGYG</a>	<a href="http://manning.com/granger/">http://manning.com/granger/</a>
<a href="http://bit.ly/3ynriE">http://bit.ly/3ynriE</a>	<a href="http://lucene.apache.org/solr/">http://lucene.apache.org/solr/</a>

从搜索的角度看，为索引添加解析后的链接可以让用户查找链接到特定网站或网页的社交媒体。想象一下，用户想要找到包含《Solr 实战》页面链接 <http://manning.com/granger/> 的所有微博。由于例子包含两个链接，需要一种方法将两个值编码在一个字段中。在 Solr 中，多于 1 个取值的文档字段称为多值字段。在 schema.xml 的字段定义中，通过设置 multiValued="true" 来声明多值字段：

```
<field name="link"
  type="string"
  indexed="true"
  stored="true"
  multiValued="true"/>
```

如果添加的文档包含多个链接，向 XML 文档添加多个 link 字段，如代码清单 5.4 所示：

代码清单 5.4 索引中 XML 文档的多值字段表示

```
<add>
  <doc>
    <field name="id">2</field>
    ...
    <field name="link">http://manning.com/granger/</field>
    <field name="link">http://lucene.apache.org/solr/</field>
    ...
  </doc>
</add>
```

在索引阶段重用相同字段名来生成多值字段。

这样就可以查询 `http://manning.com/granger/` 链接了，Solr 会对多值字段中的所有值进行匹配扫描。

截至目前，微博文档仅包含几个字段而已，很容易在 `schema.xml` 中单独声明每个字段。在现实中，并非所有文档都是如此简单或字段稀少。接下来介绍另一种字段类型——动态字段，它有助于处理规模更大更复杂的文档结构。

### 5.3.3 动态字段

Solr 的动态字段可以对文档中的一些字段赋予相同的定义，其名称匹配采用前缀或后缀模式，例如 `s_*` 或 `*_s`。动态字段使用特定的命名方案，对符合全局样式的一类字段赋以相同的字段定义。动态字段有助于解决搜索应用构建的一些常见问题：

- 多字段的文档建模
- 支持多来源的文档
- 添加新的文档来源

依次来看动态字段的这三种应用场景。首先要明确，使用 Solr 时，并不一定需要使用动态字段。如果搜索应用中没有以上任何一种情况，则完全没有必要使用动态字段。

此外，除非在开始进行文档索引时使用动态字段，否则 Solr 会在 `schema.xml` 中忽略动态字段定义。现实情况是，许多 Solr 用户只保留了 Solr 示例模式所提供的动态字段扩展列表，以供需要时使用。

#### 多字段的文档建模

动态字段使用前缀或后缀模式，在 `schema.xml` 中对匹配到的字段赋以相同的字段定义，从而实现多个字段的文档建模。如代码清单 5.3 所示，为 `type`、`screen_name` 和 `lang` 三个字段赋予字符串字段类型。此外，每个字段既要存储也要索引。除了字段名称不同，字段的定义都是完全一样的。

想象一下，除了这三个字段，还有几十个字符串字段都是既要存储也要索引。你可以对每个字段进行显示定义，或者使用一个 `<dynamicField>` 元素，在字段名称上加后缀来统一定义这些字符串字段。

```
<dynamicField name="*_s" type="string" indexed="true" stored="true" />
```

借助这个全局模式，名称以 `_s` 结尾的字段都会继承这个字段定义，例如，`subject_s`。当然，也可以使用前缀模式 `s_*`。因此，面对许多字段时，动态字段可以减少输入并简化 `schema.xml` 文件。

还可以对多值字段使用动态字段，如前一小节中的 `link` 字段。下面的动态字段定义包含 `multiValue="true"`：

```
<dynamicField name="*_ss" type="string" indexed="true" stored="true"
  multiValued="true"/>
```

对于多值 `link` 字段，XML 文档使用 `link_ss` 作为多个链接的字段名，如代码清单 5.5 所示。

#### 代码清单 5.5 索引中使用动态字段表示多值字段

```
<add>
  <doc>
    <field name="id">9999012345679</field>
    ...
    <field name="link_ss">http://manning.com/granger</field>
    <field name="link_ss">http://lucene.apache.org/solr</field>
    ...
  </doc>
</add>
```

使用动态字段命名包含多个链接。

#### 支持多来源的文档

动态字段的另一个好处是，有助于那些基础模式相同但又各自拥有独立字段的文档进行整合。当然，如果文档之间不具有共同的基础模式，那就不应该处于同一个索引中。以社交媒体为例，如果要对 Twitter、Facebook、YouTube 和 Google+ 的文档进行索引，每个来源的文档都有一些字段是代表各自所属的社交网络的。将不同来源的特定字段作为动态字段会更加直观和易于维护。举例来说，为每个来源定义多个字段的做法如下，

多个  
Facebook  
专用字段  
既存储也  
索引。

```
<field name="facebook_f1" type="string" indexed="true" stored="true" />
<field name="facebook_f2" type="string" indexed="true" stored="true" />
<field name="facebook_fn" type="string" indexed="true" stored="true" />
...
<field name="twitter_f1" type="string" indexed="true" stored="true" />
<field name="twitter_f2" type="string" indexed="true" stored="true" />
<field name="twitter_fn" type="string" indexed="true" stored="true" />
```

多个 Twitter  
专用字段既  
存储也索引。

现在只要使用一个简单的后缀为 `*_s` 模式的字符串动态字段就可以完成以上的字段定义。

```
<dynamicField name="*_s" type="string" indexed="true" stored="true" />
```

进行索引时, 需要提交 `_s` 后缀的字段, 如下代码清单 5.6 所示。

#### 代码清单 5.6 索引中使用动态字段包含特定来源的字段

```
<add>
  <doc>
    <field name="id">9999012345678</field>
    <field name="screen_name">@thelabdude</field>
    <field name="type">post</field>
    ...
    <field name="facebook_f1_s">hello</field>
    <field name="facebook_f2_s">world</field>
    <field name="twitter_f1_s">foo</field>
    <field name="twitter_f2_s">bar</field>
  </doc>
</add>
```

匹配 `*_s <dynamicField>` 定义, 这是一个字符串字段。

#### 添加新的文档来源

如果向搜索应用添加新的数据来源, 其中包含之前没有的字段, 那么可以在索引过程中通过动态字段自动添加进来。几乎每天都有新上线的社交网络, 所以我不希望不断重复修改 `schema.xml` 来处理这些新来源。通过动态字段可以不修改 `schema.xml`, 为新的文档来源添加新字段。

假如想要添加一个新的社交网站 (也许是个相亲网站), 在该网站发文时需要添加一个记录月相的字段。使用动态字段, 可以将月相字段作为字符串添加到文档中:

```
<field name="moon_phase_s">waxing crescent</field>
```

虽然动态字段对索引而言是一个很方便的功能, 但对搜索起不到什么作用。对动态字段索引的文档进行搜索时, 必须在查询中使用完整的字段名称。使用前缀或后缀模式来匹配所有字符串字段 (例如, `*_s:coffee`) 的查询是无法做到的。相反,



查询时需要明确指定字符串字段,例如,subject\_s:coffee、keyword\_s:coffee等。但是,如果需要查找多个字段,动态的或静态的,Solr 提供复制字段来处理这种情况。

### 5.3.4 复制字段

Solr 的复制字段允许将一个或多个字段值填充到一个字段中。具体来说,在大多数搜索应用中复制字段支持以下两种情况:

- 将多个字段内容填充到一个字段。
- 对同一字段内容进行不同的文本分析,创建一个新的可搜索字段。

#### 创建多字段的汇聚字段

绝大多数搜索应用都只提供一个搜索框,供用户输入查询内容。这样做的目的是,帮助用户快速查找文档,而无须填写复杂的表单。想想看,一个简单的搜索框成就了 Google。在微博搜索示例中,你可能会觉得只要能搜索微博文本就大功告成了,这也太简单了。但如果用户搜索 @thelabdude, 由于它包含在 screen\_name 字段中,而不是 text 字段中,那么使用一个搜索框就无法找到想要的结果。另外,如果微博包含短网址(如 bit.ly 这类),由于它们都存储在 links 字段中,所以搜索 text 字段时,已解析的网址将不会被匹配出来。所以,我们想要一个将 screen\_name、text 和已解析的 link 字段汇聚在一起的字段。所幸,在 Solr 中很容易做到这一点,使用 <copyField> 可以将文档中的多个字段汇聚成一个搜索字段。

首先,需要定义一个让其他字段能复制到一起的目标字段,这里命名为 catch\_all 字段:

```
<field name="catch_all"
  type="text_en"
  indexed="true"
  stored="false"
  multiValued="true"/>
```

由于 catch\_all 字段来自于其他字段,所以它不应被存储。

如果来源字段是多值字段,则目标字段也必须是多值字段。

这看起来与其他字段没什么差别,不过有两点需要注意。

首先,该字段不存储(stored="false"),这样做是有一定道理的。我们不希望向用户显示多个字段连接在一起的一堆东西。事实上,即使想这样做也无法做到,Solr 不能返回复制字段的原始值。请记住,Solr 能返回存储字段的原始值。

其次,如果来源字段是多值字段,目标字段必须设置为多值字段(multiValued="true")。在微博搜索示例中,link 字段是多值字段,因此复制字段也必须是多值字段。此外,如果从多个来源字段复制到目标字段,即便所有的来源字段是单值字段,目标字段也必须设置为多值字段。

现在已经定义好了目标字段，需要使用 `<copyField>` 来明确 Solr 要从哪些字段进行复制。代码清单 5.7 展示了如何使用 `<copyField>` 从 `screen_name`、`text` 和 `link` 字段中复制字段值到 `catch_all` 字段：

代码清单 5.7 使用 `<copyField>` 填充 `catch_all` 字段

```
<schema>
  <fields>
    ...
  </fields>
  <copyField source="screen_name" dest="catch_all" />
  <copyField source="text" dest="catch_all" />
  <copyField source="link" dest="catch_all" />
</types>
...
</types>
</schema>
```

screen\_name、text 和 link 字段的内容被全部复制到 catch\_all 字段。

需要注意的是，在 `schema.xml` 中 `<copyField>` 元素是 `<fields>` 和 `<type>` 的同级元素。想想看为什么是这样？首先必须定义来源（source）和目标（dest）字段。当所有字段都定义好之后，通过 `copyField` 将它们连接起来。这是作为同级元素的最佳解释。

### 对一个字段应用不同的分析器

你可能希望对一个字段的内容进行不同的分析。第 6 章将介绍词干提取技术，即把词项转换成通用的基础形式，也就是词干，以提高查全率（详见第 3 章）。通过词干提取，`fishing`、`fished` 与 `fishes` 都会归一为 `fish` 这个词干。因此，词干提取让用户在搜索文档时不用考虑一个词的多种可能语言形式，这是一般文本搜索字段的好方法。

考虑一下词干提取如何影响搜索输入时的建议框（自动建议）。在这种情况下，词干提取仅能给用户建议词干本身，而不能建议完整的词项。例如，启用词干提取后，当用户开始输入 `human` 时，自动建议框无法给出 `humane` 或 `humanities` 这样的建议。Solr 的复制字段具有一定灵活性，可选择启用或停用特定的文本分析功能（如词干提取），无须在索引中重复存储。代码清单 5.8 取自 `schema.xml`。

代码清单 5.8 在同一文本上使用 `copyField` 实现不同文本分析功能

```
<field name="text"
  type="stemmed_text"
  indexed="true"
  stored="true"/>
<field name="auto_suggest"
  type="unstemmed_text"
```

```
indexed="true"
stored="false"
multiValued="true"/>
...
```

```
<copyField source="text" dest="auto_suggest" />
```

将 text 字段的原始文本内容复制到 auto\_suggest 字段，使用 unstemmed\_text 字段类型进行分析。

在这种情况下，text 字段是 stemmed\_text 字段类型，也就是说这个字段中的文本是经过词干提取的。auto\_suggest 字段没有经过词干提取。我们使用 <copyField> 将 text 字段未经词干提取的文本赋予 auto\_suggest 字段。在系统后端，Solr 将 text 字段原生的、未被分析的内容发送给 auto\_suggest 字段，以便于采取不同的文本分析策略。原始文本只存储一次。需要重申的是，搜索结果中不能返回 auto\_suggest 字段的原始值，因此它的属性是 stored="false"。第 10 章会介绍如何实现自动建议功能。

### 5.3.5 唯一键字段

5.2.2 节中讨论过如何在索引中使用唯一的 ID 值来识别文档，这是一个好主意。重申一下，如果每个文档都赋予一个唯一标识符字段，Solr 将在索引创建时避免产生重复索引。此外，如果计划将 Solr 的索引分发到多台服务器上，那就必须为文档提供唯一标识符。鉴于以上原因，建议从一开始就为文档定义唯一标识符。在微博搜索示例中，id 字段是独一无二的，因此对 Solr 进行配置时，我们在 schema.xml 中使用 <uniqueKey> 元素，将 id 字段作为文档的唯一键。

#### 代码清单 5.9 <uniqueKey> 元素表示文档的唯一 ID 字段

```
<uniqueKey>id</uniqueKey>
```

需要注意一点，这里最好使用基本字段类型，如字符串型或长整型。对确定为 <uniqueKey/> 的字段，要确保 Solr 在索引时不会修改该字段值。在之前的例子中我们已经看到，如果不使用字符串作为文本型键值的类型，Solr 就不能正确地返回结果。为减少麻烦，唯一键字段应使用字符串类型或其他基本字段类型。

至此，我们已经介绍了在 schema.xml 中进行字段定义的基本知识。你也应该已经掌握了如何使用多值字段、动态字段和复制字段。现在是时候进入 Solr 的 schema.xml 的下一个主要部分，学习如何定义字段类型。

## 5.4 结构化非文本字段类型

本节介绍如何定义结构化数据（如日期、语言代码和用户名）的字段类型。第

6 章将介绍类似微博搜索示例中正文文本字段的类型定义方法。通常 Solr 为结构化数据内建了许多字段类型，例如，数字、日期和地理位置字段等。图 5.4 是 Solr 中一些最常用字段类型的类图。

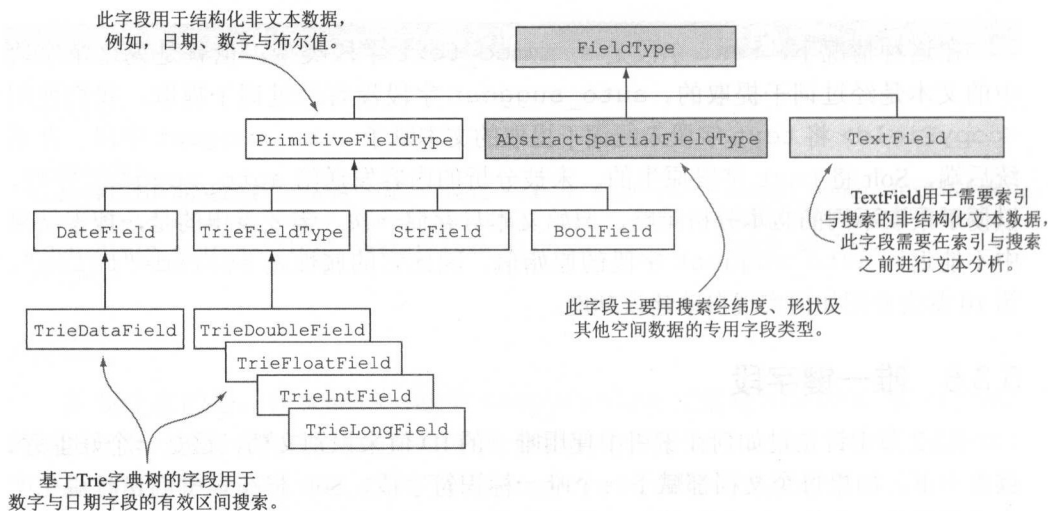


图 5.4 Java 包 org.apache.solr.schema 中最常用的字段类型类图

我们从最常见的字符串 string 字段类型开始讨论非文本数据的字段类型。

### 5.4.1 字符串字段

在前面的微博例子中，除了 text 字段之外，screen\_name、type、timestamp 与 lang 都应该是索引字段。那么，就需要为这四个字段确定合适的字段类型。事实上，每个字段包含的结构化数据都不需要进行分析。例如，lang 字段包含 ISO-639-1 标准的语言代码，用来识别微博的语种，例如，en 表示英语。用户通过搜索 lang 字段，可以找到英文微博，如图 5.5 所示。

因为语言代码已经标准化，所以 Solr 在索引与查询处理时无须进行任何更改。Solr 为那些不需要变更的结构化值提供 string 字段类型。下面的代码清单演示了在 schema.xml 中如何定义 string 字段类型。

#### 代码清单 5.10 schema.xml 中的 string 字段类型定义

```
<fieldType name="string" class="solr.StrField"
  sortMissingLast="true" omitNorms="true"/>
```

技术背后是，所有的字段类型由一个 Java 类来实现，这里使用 solr.StrField。在执行阶段，solr.StrField 解析为 Solr 的内置类 org.apache.

`solr.schema.StrField`。任何时候，你在 `schema.xml` 中都会看到 `solr.` 作为类的前缀，其对应的完整 Java 包是 `org.apache.solr.schema`。这种简化标记有助于保持 `schema.xml` 的内容整齐。`sortMissingLast` 和 `omitNorms` 属性是高级选项，5.4.4 节会详细介绍。

如果将 `lang` 字段设置为字符串类型，Solr 会把文档的取值 `en` 存储在索引中，并作为不变的持久索引。查询时需要传递准确值 `en`，才能匹配出英文文档。图 5.5 中用户选择了英语，在表单处理时，需要转换为 `en`。`<fieldType>` 字符串也适用于 `screen_name` 字段和 `type` 字段，但 `timestamp` 时间戳字段怎么办呢？

The screenshot shows a web browser window titled "Microblog Search Tool" with the URL `http://example.com/SolrInAction/TextAnalysis`. The main form is titled "Social Media Search Form" and contains the following fields:

- By User:** A text input field containing `@thelabdude`.
- Type:** A dropdown menu with options `Post`, `Reply`, and `Retweet`.
- Language:** A dropdown menu with the option `English`.
- Date Range:** A dropdown menu with the option `After` and a date input field containing `05/01/2012`.
- With Text:** A text input field containing `San Francisco north beach coffee`.

A **Search** button is located below the form. To the right of the form, there are labels with arrows pointing to the corresponding fields: "搜索微博的非文本字段:" (Non-text fields for searching Weibo), "用户名" (Username), "类型" (Type), "语种" (Language), and "时间戳" (Timestamp).

Below the form, the **Search Results** section displays the following information:

- `@thelabdude` on May 22, 2012 @ 09:30 AM
- ♥ Favorited by 10 users
- `#Yummm :) Drinking a latte at Caffé Grecco in SF's historic North Beach... Learning text analysis with #SolrInAction by @ManningBooks on my i-Pad`

图 5.5 使用结构化非文本字段的微博搜索表单

## 5.4.2 日期字段

日期字段搜索的常见做法是允许用户指定日期范围。图 5.5 中用户搜索了某一特定日期（05/01/2012）之后发布的微博。在查询端，`timestamp` 字段上执行的是一个区间查询：

```
timestamp:[2012-05-01T00:00:00Z TO *]
```

由于搜索日期区间是常见的用例，Solr 提供了一个优化的内置 `<fieldType>`，

称为 tdate, 如下所示:

代码清单 5.11 schema.xml 中的日期字段类型定义

```
<fieldType name="tdate" class="solr.TrieDateField" omitNorms="true"
  precisionStep="6" positionIncrementGap="0"/>
```

说实话, 这看起来比字符串类型更唬人! 像 precisionStep 和 positionIncrementGap 这样的附加属性都是高级选项, 5.4.4 节会介绍。再次强调, solr.TrieDateField 是 Solr 的简化标记, 用于代表此字段类型实现的 Java 类名, 即 org.apache.solr.schema.TrieDateField。trie 是一种高级的基于树的数据结构, 可以对数值和日期值进行不同精确程度的有效搜索。

Solr 在索引时需要知道如何解析日期。回想 5.1 节的示例 XML 文档, 我们将它提交给 Solr 进行索引, 包含了以下 timestamp 字段:

```
<add>
  <doc>
    ...
    <field name="timestamp">2012-05-22T09:30:22Z</field>
    ...
  </doc>
</add>
```

在 schema.xml 中 timestamp 字段定义为 tdate 类型:

```
<field name="timestamp" type="tdate" indexed="true" stored="true" />
```

通常情况下, Solr 默认采用 ISO-8601 日期/时间格式 (YYYY-MM-DDTHH:MM:SSZ), 微博的日期 (2012-05-22T09:30:22Z) 分解如下:

```
yyyy = 2012
mm = 05
dd = 22
HH = 09 (24-hr clock)
mm = 30
ss = 22
Z = UTC Timezone (Z is for Zulu)
```

如果向 Solr 提交其他日期格式, 索引时就会得到验证错误信息, 待索引的文档会被拒绝。

### 日期粒度

接下来需要确定索引中的日期粒度。这又要回到实际需求, 了解用户如何进行日期查询。如果用户只希望按照天来查询, 那就不需要对日期索引精确到秒或毫秒。

如果需要按日期对文档排序，那么小时级别的粒度可能就太粗了，这时候需要达到分钟级别的粒度。

在索引创建阶段，Solr 支持日期的数学运算，可以帮助你毫不费力地达到日期字段准确的精确度。比方说，你决定需要在小时级别的粒度上索引日期字段，这样可以节省索引空间，同时也意味着，用户搜索时无法在小时区间内指定更精确的搜索。当索引时，日期以 /HOUR 结尾，/ 告诉 Solr 将日期“四舍五入”处理成小时级别。下面一起来看看如何将示例微博按照小时级别的粒度进行索引。

```
<field name="timestamp">2012-05-22T09:30:22Z/HOUR</field>
```

当索引时，示例文档的时间戳字段值等价于 2012-05-22T09:00:00Z。除了指定准确的日期/时间之外，Solr 还支持 NOW 关键字，它表示 Solr 服务器的当前系统时间。运用 Solr 的日期数学运算将具体日期或 NOW 关键词结合起来，可以实现强大的日期计算。例如，NOW/DAY 表示当前日的午夜，NOW/DAY+1DAY 表示明天的午夜。要查询自今天起的所有文档，设置时间戳为：[NOW/DAY TO NOW/DAY + 1DAY]。第 7 章会介绍更多的区间查询。

提醒一下，需要进行日期区间查询时，tdate 字段是不错的选择，但这样会需要更多的索引空间，因为每个日期值被分解为更多的时间单元进行存储。参考 Solr 的 Java 文档，precisionStep = "6" 对长字段来说比较适合。5.4.4 节会介绍 precisionStep 的正确选择方法。

### 5.4.3 数值字段

大多数情况下 Solr 对数值字段的处理方式与你所期望的相同。5.1 节讨论了如何用 favorites\_count 字段表示其他用户对微博点赞的次数。这不是一个可以直接用于搜索的字段，但从显示与排序角度看它是非常有用的。换句话说，假设用户想要根据 favorites\_count 字段对最热门作者的微博进行排序，在 schema.xml 中定义字段如下：

```
<field name="favorites_count" type="int" indexed="true" stored="true" />
```

int 字段类型定义如下

```
<fieldType name="int" class="solr.TrieIntField"
  precisionStep="0" positionIncrementGap="0"/>
```

由于不需要在这个字段上进行区间查询，因此设定 precisionStep="0"。这是最佳的排序设定，不会带来额外的存储成本，而与之相关的是，更快的区间查询需要更高的精确梯度。另外，不应采用字符串字段排序那样对数值字段进行索



引。如果字段类型是基于字符串的，那么 Solr 会进行词汇排序，而不是按数值排序。换句话说，如果使用基于字符串的字段类型对数值字段进行索引，排序结果会像 1,10,2,3, ...，而不是 1,2,3,... 10。

至此，我们已经讨论了包含结构化信息的索引字段的主要概念。后续章节会介绍这些非文本字段类型的具体应用。例如，第 15 章 Solr 的地理空间搜索使用 <fieldType> 来表示经度与纬度。接下来简要介绍字段类型的高级配置选项来结束这一小节。

5.4.4 高级字段类型属性

Solr 支持字段类型的可选属性，通过可选属性启用高级功能。表 5.3 涵盖了 <fieldType> 元素的高级属性。

表 5.3 schema.xml 中 fieldType 元素的高级属性概览

属性	启用后的行为 (="true")
sortMissingFirst	排序结果时，Solr 会列出那些排在搜索结果最前面的，但该字段却没有取值的文档
sortMissingLast	排序结果时，Solr 会列出那些排在搜索结果最后面的，但该字段没有取值的文档
precisionStep	在基于 trie 的字段（如 TrieDate 与 TrieLong）上执行快速地区间查询，确定索引中数值表示所需的索引项数量；precisionStep 属性的更多信息，参见 JavaDoc 的 NumericRangeQuery 类
positionIncrementGap	用于防止短语查询在多值字段中匹配到一个值的结尾和下一个值的开头

Solr 新手常常对 precisionStep 比较困惑，接下来会详细介绍它。不过当前你可以跳过下面的讨论，当部署好搜索应用后再回过头来看，寻求排序和区间查询的性能优化方法。

为数值字段选择最佳 precisionStep

找寻文档时经常会碰到两种情况：对数值或日期字段进行区间匹配，称为区间查询；按照数值和日期字段进行结果排序。之前提到过，Solr 使用字典树（trie）数据结构来支持有效的区间查询，对数值和日期进行排序。接下来介绍如何选择 precisionStep 最佳值来支持 Solr 搜索应用中的区间查询与排序。

首先，当用户在数值或日期字段上通过区间搜索查找文档时，索引中是否存在数值或日期字段，然后再考虑 precisionStep 的问题。对于每个字段，考虑索引时可能的区间值，唯一值数量是潜在的百万级还是只有少数一些？在 Solr 中，字段唯一值的数量称为字段的基数（cardinality）。

例如, 在全美房屋出售信息的 Solr 索引上进行搜索, 购房者通常会在特定区域和价格区间上搜索房屋。搜索应用中的典型查询可能是这样的: `city:Denver AND price:[250000 TO 300000]`。房屋价格是典型的需要进行有效区间搜索的字段。由于这个搜索应用涵盖了美国各地的房屋信息, 所以价格字段的取值从 10 000 美元到 1 000 万美元不等, 因此价格区间的基数很大。

接下来, 需要确定价格显示的最佳字段类型。一般而言, 大多数的房屋价格会以美元标价, 很少有房屋价格精确到美分。因此, `int` 或 `long` 字段应该足够了。此外, Solr 的最大整数值为 2 147 483 647 (21 亿), 事实上不可能有房屋价格超过这个上限。始终牢记, 要尽可能节约地使用字段类型, 也就是说, 应避免出现浪费。例如, 如果 4 个字节的整数型就够了, 却使用了 8 个字节的长整型的情况。这样做会减少磁盘上的索引大小, 并在搜索与排序时降低内存使用。房屋价格字段定义如下:

```
<field name="listing_price" type="tint" indexed="true" stored="true" />
```

为了支持区间查询, 该字段必须进行索引。由于要在搜索结果中显示房屋价格, 所以该字段也必须被存储。在 Solr 的示例 `schema.xml` 中, `tint` 字段类型定义如下:

```
<fieldType name="tint" class="solr.TrieIntField"
  precisionStep="8"
  positionIncrementGap="0"/>
```

使用 `TrieInt` 字段, 属性 `precisionStep="8"`, 那么 327 500 美元的房屋价格怎样索引? 基于字典树的字段背后, Lucene 会对字段里的每个值产生多个索引项, 每个索引项的精确度都低于原始值。换句话说, 两个不同的房屋价格在低精确度下会有重叠。Lucene 这种做法是为了减少区间查询匹配中的索引项数量。表 5.4 列举了房屋价格为 327 500 美元、`precisionStep="8"` 被索引后的索引项。

表 5.4 对房屋挂牌价 327 500 美元使用 `TrieIntfield` 及 `precisionStep` 为 8 的索引项

精确步长 (8)	运算	索引项
0: 没有位移除	327500 & 0xFFFFFFFF	327500
1: 8 个最低有效位被移除	327500 & 0xFFFFF00	327424
2: 16 个最低有效位被移除	327500 & 0xFFFF0000	262144
3: 24 个最低有效位被移除	327500 & 0xFF000000	0

表 5.4 显示了不同的精确度步长。Lucene 以 8 个字节为一个步长, 从原始值中移除相应的字节数, 得到的索引项精度逐步降低。例如, 步长 2 中房屋价格 327 500 美元和 326 800 美元的索引项都是 262 144, 这意味着区间查询中这两个房屋价格只需匹配 262 144 即可。事实上, 262 144 这个索引项是房屋价格区间在 262 144 美元

和 327 678 美元内所有价格对应的索引项。换句话说,precisionStep 为 8 的设定,允许 Solr 对 262 144 美元与 327 679 美元之间的所有房屋价格使用 262 144 这个索引项。

设定房屋价格的precisionStep = "4",比较一下更小梯度带来的影响。表 5.5 列举了 precisionStep="4" 的房屋价格 327 500 的索引项。

表 5.5 对房屋价格 327 500 美元使用 TrieInt 字段及 precisionStep 为 4 的索引项

精确步长 (4)	Java 位运算	索引项
0: 没有位移除	327500 & 0xFFFFFFFF	327500
1-4 个最低有效位被移除	327500 & 0xFFFFFFFF0	327488
2-8 个最低有效位被移除	327500 & 0xFFFFFFFF00	327424
3-12 个最低有效位被移除	327500 & 0xFFFFFFFF000	323584
4-16 个最低有效位被移除	327500 & 0xFFFF0000	262144
5-20 个最低有效位被移除	327500 & 0xFFF00000	0

表 5.5 显示,精确步长越小产生的索引项越多,精确步长为 4 产生 6 个索引项,而精确步长为 8 产生 4 个索引项。一般情况下,每个字段值使用更小的精确步长时会产生更多的索引项,因此会增加索引大小。但是, Lucene 在更多索引项中会更快地缩小搜索范围,因此更多的索引项会提高区间查询速度。直接的解释是, Lucene 使用字典树中最低的可能精确度作为搜索区间的中心。然而,由于必须对搜索区间的上限与下限进行更精确地搜索,所以更多的索引项让区间边界匹配更加有效。

为了验证这一点,我们做一次非正式的基准测试。在房屋价格 110 000 美元与 5 000 000 美元之间随机索引 500 000。索引完成后,随机生成 10 000 个区间查询来体验平均查询性能。表 5.6 是测试结果总结。

表 5.6 对基于字典树的房屋价格字段进行 precisionStep 为 4 和 8 的非正式基准测试结果,比较分析了索引大小、索引项数量与查询性能

精确步长	索引项数量	索引大小 (KB)	区间查询性能
8	68 074	28 612	7.0 ms
4	118 170	32 496	6.3 ms

请注意,每个文档的索引大小相差大约 8 个字节,这与表 5.5 的情况保持一致,当使用更小的精确步长时,每个文档生成了两个额外的 4 字节整数索引项。总之,选择精确步长时,必须对区间查询性能与索引空间进行权衡。房屋价格字段定义为 TrieInt 类型,精确步长为 4 时,对特定价格的查询会产生更多索引项,但区间搜索速度更快,特别是在字段的基数很大并包含许多唯一值的情况下。

## 5.5 发送文档到Solr进行索引

现在我们已经有足够的索引处理知识，接下来将文档添加到 Solr。本节讲解如何将文档发送到 Solr 进行索引，并简单介绍背后原理。学习完本节，你就可以为搜索应用添加文档索引了。以本章示例微博为例介绍如何进行索引。

### 5.5.1 使用 XML 或 JSON 进行文档索引

正如 5.1 节谈到的，Solr 使用简单的 XML 文档结构来添加文档。代码清单 5.12 展示了本章之前使用的两个示例微博。在这里，我们更改了字段名称，以便使用动态字段。例如，由于字段名称里有 `_s` 后缀，所以 `screen_name_s` 是字符串类型。这样做只是为了图方便，当然也可以对 `schema.xml` 不做任何更改，将这些文档直接添加到 Solr 示例服务器。如果正在构建真实的搜索应用，那么需要在 `schema.xml` 中显式声明代码清单 5.3 中的字段。在当前这个示例中使用动态字段比较好。

代码清单 5.12 在 Solr 中使用动态字段，用来索引示例微博的 XML 文档

告诉 Solr，我们要向索引添加新的文档。

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="screen_name_s">@thelabdude</field>
    <field name="type_s">post</field>
    <field name="lang_s">en</field>
    <field name="timestamp_tdt">2012-05-22T09:30:22Z/HOUR</field>
    <field name="favorites_count_ti">10</field>
    <field name="text_t">#Yummm :) Drinking a latte at Caffe Grecco in SF's
      historic North Beach... Learning text analysis with #SolrInAction
      by @ManningBooks on my i-Pad</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="screen_name_s">@thelabdude</field>
    <field name="type_s">post</field>
    <field name="lang_s">en</field>
    <field name="timestamp_tdt">2012-05-23T09:30:22Z/HOUR</field>
    <field name="favorites_count_ti">10</field>
    <field name="text_t">Just downloaded the ebook of #SolrInAction from
      @ManningBooks http://bit.ly/T3eGYG to learn more about #Solr
      http://bit.ly/3ynriE</field>
    <field name="link_ss">http://manning.com/grainger/</field>
    <field name="link_ss">http://lucene.apache.org/solr/</field>
  </doc>
</add>
```

一次可以增加多个文档，每个文档用 `<doc>` 标签封装。

通过字段名称前缀，使用动态字段确定每个字段的字段类型。

将这个 XML 文档发送给 Solr，生成了两条微博索引。Solr 示例包含一个简单的命令程序，负责将 XML 文档发送到示例服务器。打开工作站上的命令行窗口，

输入代码清单 5.13 中的命令：

代码清单 5.13 在 Solr 中索引示例微博的命令

```
cd $SOLR_IN_ACTION/example-docs/
java -jar post.jar ch5/tweets.xml

SimplePostTool: version 1.5
SimplePostTool: POSTing files to http://localhost:8983/solr/update..
SimplePostTool: POSTing file ch5/tweets.xml
SimplePostTool: COMMITting Solr index changes..
```

使用实际路径替换 SOLR\_IN\_ACTION。

post.jar 应用的输出。

提交这些文档，让它们可以被搜索到。

此时，在示例 Solr 服务器上应该索引了两条示例微博。验证一下索引是否成功：打开常用的 Web 浏览器，在地址栏输入 <http://localhost:8983/solr/#/>，进入 Solr 管理面板，点击左侧菜单中 collection1 下的 Query 链接，执行查询 type\_s:post，如图 5.6 所示。

Click collection1 (Collection1) below the query (Query).

Use type\_s:post to view the tweet submitted by post.jar.

Search results returned two example tweets.

Request-Handler (qt) /select

q type\_s:post

fq

sort

start, rows 10

Raw Query Parameters key1=val1&key2=val2

wt xml

☒ indent

☐ debugQuery

☐ dismax

☐ edismax

☐ hl

☐ facet

☐ spatial

☐ spellcheck

Execute Query

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
    <lst name="params">
      <str name="indent">true</str>
      <str name="q">type_s:post</str>
      <str name="wt">1383080401664</str>
      <str name="wt">xml</str>
    </lst>
  </lst>
  <result name="response" numFound="2" start="0">
    <doc>
      <str name="id">1</str>
      <str name="screen_name_s">@thelabduke</str>
      <str name="type_s">post</str>
      <str name="lang_s">en</str>
      <date name="timestamp_tdt">2012-05-22T09:00:00Z</date>
      <int name="favourites_count_ti">10</int>
      <str name="text_t">Yummm :) Drinking a latte at Caffe Grecco in SF's histor<
      <long name="_version_">1450264911258058752</long></doc>
    <doc>
      <str name="id">2</str>
      <str name="screen_name_s">@thelabduke</str>
      <str name="type_s">post</str>
      <str name="lang_s">en</str>
      <date name="timestamp_tdt">2012-05-23T09:00:00Z</date>
      <int name="favourites_count_ti">10</int>
      <str name="text_t">Just downloaded the ebook of #SolrInAction from @Manning<
      <str name="link_ss">
        <str>http://manning.com/grainger/</str>
        <str>http://lucene.apache.org/solr/</str>
      </str>
      <long name="_version_">1450264911261204480</long></doc>
    </result>
  </response>
```

图 5.6 使用 Solr 示例提供的 post.jar 命令行工具添加示例微博，通过 type\_s:post 查看索引结果

从技术上讲,借助 post.jar 这个工具,通过 HTTP 将 XML 文档发送给 Solr 的更新处理器 (<http://localhost:8983/solr/collection1/update>)。更新处理器支持文档的添加、更新与删除。5.6 节会详细介绍更新处理器。

除了 XML 以外, Solr 的更新请求处理器还支持流行的 JSON 和 CSV 文档格式。示例微博 XML 文档的等价 JSON 文档格式如代码清单 5.14 所示。

代码清单 5.14 不使用 XML, 而使用 JSON 索引文档

```
[
{
  "id" : "1",
  "screen_name_s" : "@thelabdude",
  "type_s" : "post",
  "lang_s" : "en",
  "timestamp_tdt" : "2012-05-22T09:30:22Z/HOUR",
  "favorites_count_ti" : "10",
  "text_t" : "#Yummm :) Drinking a latte at Caffe Grecco in SF's historic
    North Beach... Learning text analysis with #SolrInAction by
    @ManningBooks on my i-Pad"
},
{
  "id" : "2",
  "screen_name_s" : "@thelabdude",
  "type_s" : "post",
  "lang_s" : "en",
  "timestamp_tdt" : "2012-05-23T09:30:22Z/HOUR",
  "favorites_count_ti" : "10",
  "text_t" : "Just downloaded the ebook of #SolrInAction from
    @ManningBooks http://bit.ly/T3eGYG to learn more about
    #Solr http://bit.ly/3ynriE",
  "link_ss" : [ "http://manning.com/grainger/",
    "http://lucene.apache.org/solr/" ]
}
]
```

← 将待索引的所有文档封装成一个 JSON 数组。

← 索引的每个文档是一个 JSON 对象。

← 多值字段编码为一个 JSON 数组。

同样地, 可以使用 post.jar 工具将 JSON 文档发送给 Solr, 但 XML 是默认文档格式。因此, 必须在搜索应用中设定 type 的系统属性为 application/json, 以指明发送的是 JSON。

```
java -Dtype=application/json -jar post.jar ch5/tweets.json
```

命令行中 -Dtype=application/json 参数的位置非常重要, 必须位于 -jar 参数之前。要了解 post.jar 支持的完整选项列表, 运行 java -jar post.jar --help 进行查看。

如果已经做完 XML 和 JSON 文档的添加练习，你可能会认为索引中现在已经有 4 个文件。然而，由于文档中使用了 `id` 字段，所以索引里只有 2 个文档。为了明确这一点，你可以重新提交 `type_s:post` 来验证。这也说明 Solr 如何使用唯一键字段对已有文档进行更新，示例 `schema.xml` 中的唯一键是 `id`。

进一步仔细查看 `post.jar` 工具的输出，你会发现：当发送文档之后，`post.jar` 向 Solr 发送了一个提交命令。不管如何将文档发送到 Solr，在提交之前它们都无法被搜索。提交过程相当复杂，5.6 节会详细介绍。下一小节介绍如何使用 SolrJ（一个流行的 Java 客户端）来索引文档。

## 5.5.2 使用 SolrJ 客户端库添加文档索引

SolrJ 是一个由 Solr 项目提供的基于 Java 的客户端库，用于 Java 程序与 Solr 服务器进行通信。本节将编写一个简单的 SolrJ 客户端，使用 Java 发送文档。如果你不熟悉 Java 开发，或者搜索应用不是使用 Java 编写的，那么可以选择其他编程语言版本的 Solr 客户端库，如 .NET、Ruby、Python 和 PHP。第 12 章（12.8.2 节）会介绍各种编程语言的 Solr 客户端库。

代码清单 5.15 是使用 SolrJ 添加两个示例微博文档进行索引的简单例子，执行的是硬提交。提交之后，示例代码向 Solr 发送一个匹配所有文档的查询（`*:*`），在搜索结果中返回已经索引的文档。

代码清单 5.15 SolrJ 客户端应用程序举例

```
package sia.ch5;
...
import org.apache.solr.client.solrj.SolrQuery;
import org.apache.solr.client.solrj.SolrServer;
import org.apache.solr.client.solrj.SolrServerException;
import org.apache.solr.client.solrj.impl.HttpSolrServer;
import org.apache.solr.client.solrj.response.QueryResponse;
import org.apache.solr.common.SolrDocument;
import org.apache.solr.common.SolrDocumentList;
import org.apache.solr.common.SolrInputDocument;

public class ExampleSolrJClient {

    public static void main(String[] args) throws Exception {
        String serverUrl = (args != null && args.length > 0) ? args[0] :
            "http://localhost:8983/solr/collection1";
```



```
SolrServer solr = new HttpSolrServer(serverUrl);
```

通过指定 URL (<http://localhost:8983/solr/collection1>) 连接到 Solr 服务器。

```
SolrInputDocument doc1 = new SolrInputDocument();
doc1.setField("id", "1");
doc1.setField("screen_name_s", "@thelabdude");
doc1.setField("type_s", "post");
doc1.setField("lang_s", "en");
doc1.setField("timestamp_tdt", "2012-05-22T09:30:22Z/HOUR");
doc1.setField("favorites_count_ti", "10");
doc1.setField("text_t", "#Yummm :) Drinking a latte at Caffe Grecco"
    in SF's historic North Beach... Learning text analysis with
    #SolrInAction by @ManningBooks on my i-Pad");

solr.add(doc1);
```

通过 HTTP 发送 SolrInputDocument 到 Solr 更新请求处理器。

```
SolrInputDocument doc2 = new SolrInputDocument();
doc2.setField("id", "2");
doc2.setField("screen_name_s", "@thelabdude");
doc2.setField("type_s", "post");
doc2.setField("lang_s", "en");
doc2.setField("timestamp_tdt", "2012-05-22T09:30:22Z/HOUR");
doc2.setField("favorites_count_ti", "10");
doc2.setField("text_t", "Just downloaded the ebook of #SolrInAction
    from @ManningBooks http://bit.ly/T3eGYG to learn more about #Solr
    http://bit.ly/3ynriE");
doc2.addField("link_ss", "http://manning.com/granger/");
doc2.addField("link_ss", "http://lucene.apache.org/solr/");
```

使用 addField 方法，为多值字段增加多个字段值。

```
solr.add(doc2);
solr.commit(true, true);

for (SolrDocument next : simpleSolrQuery(solr, ".*", 10)) {
    prettyPrint(System.out, next);
}
```

```
static SolrDocumentList simpleSolrQuery(SolrServer solr,
    String query, int rows) throws SolrServerException {
    SolrQuery solrQuery = new SolrQuery(query);
    solrQuery.setRows(rows);
    QueryResponse resp = solr.query(solrQuery);
    SolrDocumentList hits = resp.getResults();
    return hits;
}
```

使用 SolrQuery 对象构造出匹配所有文档的查询。

```
static void prettyPrint(PrintStream out, SolrDocument doc) {
    List<String> sortedFieldNames =
        new ArrayList<String>(doc.getFieldNames());
    Collections.sort(sortedFieldNames);
    out.println();
    for (String field : sortedFieldNames) {
        out.println(String.format("\t%s: %s",
            field, doc.getFieldValue(field)));
    }
    out.println();
}
```

以可读方式打印输出 (prettyPrint) 搜索结果的每个 SolrDocument。

使用 SolrInputDocument 对象构造一个被索引的文档。

执行正常的 (硬) 提交, 确保新索引的文档可以被搜索到。

从这个基础例子可以看出, SolrJ 的 API 可以很容易地连接到 Solr, 进行文档添加、查询发送与搜索结果处理等操作。首先需要 Solr 服务器的地址, 示例中的地址是 `http://localhost:8983/solr/collection1`。背后使用的技术是, SolrJ 使用 Apache 的 `HttpComponents` 客户端库, 通过 HTTP 与 Solr 服务进行通信。在 5.5.1 节中已知 Solr 支持 XML 和 JSON, 因此需要知道 SolrJ 使用其中哪一种格式与 Solr 进行连接。事实上, SolrJ 默认使用 `javabin` 作为内部二进制协议。当进行 Java-to-Java 的通信时, `javabin` 协议比使用 XML 或 JSON 效率更高。

除了向单台 Solr 服务器发送请求之外, SolrJ 内置了文档批处理机制, 以支持大规模索引、Solr 实例间的负载平衡、SolrCloud 配置中 Solr 服务器位置的自动发现, 以及将 Solr 作为非服务器模式内嵌在 Java 应用中。第 12 章会详细地介绍这些内容。

### 5.5.3 向 Solr 导入文档的其他工具

我们已经学会了向 Solr 导入文档的两种基本方法: 使用 HTTP POST 方式的 `post.jar` 应用和使用 Java 编程的 SolrJ 客户端。但这不是向 Solr 导入的全部方法。作为成熟的、广泛应用的开源技术, Solr 提供了许多功能强大的工具, 用于从其他系统中添加文档。本节介绍 3 种流行的文档索引导入工具:

- 数据导入处理器 (Data Import Handler, DIH)
- `ExtractingRequestHandler`, 又名 Solr Cell
- Nutch

上述每个工具都很强大, 如果展开来介绍都需要一整章的篇幅。此处只简要介绍这些工具, 现在只要知道有些文档索引的导入方法即可。

#### 数据导入处理器

数据导入处理器 (DIH) 是从一个或多个外部来源 (网站或关系型数据库) 将数据导入到 Solr 的一个扩展。DIH 可以与具备主流 JDBC 驱动的任何数据库对话, 例如, Oracle、Postgres、MySQL 及 MS SQL Server。概括而言, 将数据库连接参数和 SQL 查询语句提供给 Solr, DIH 组件会对数据库进行查询, 将查询结果转换为 Solr 索引所需的文档。另外, 还可以轻松地从 XML 文件或外部网站获取数据, 作为待索引的文档。第 12 章会更详细地介绍 DIH, 附录 C 提供了外部数据导入的映射处理。现在来看看另一个处理二进制文件 (如 PDF 与 Word 文档) 的索引工具。

#### `ExtractingRequestHandler`

`ExtractingRequestHandler`, 又名 Solr Cell, 可以抽取二进制文件中的文本 (如 PDF、微软 Office 和 OpenOffice 等) 内容进行索引。这背后的技术是, Solr Cell 使用 Apache 的 Tika 工具进行文本抽取。具体来说, Tika 提

供组件来检测文档类型，并从二进制文件中抽取文本和元数据。例如，向 `ExtractingRequestHandler` 发送一个 PDF 文件，它会自动在 Solr 索引中生成 `title`（标题）、`subject`（主题）、`keywords`（关键词）与 `text`（正文文本）这样的索引字段。本书不会涉及很多有关 `ExtractingRequestHandler` 的配置说明，第 12 章会再介绍一些，通过一个完整的教程将二进制文件索引到 Solr 中。

## Nutch

Apache 的 Nutch 是一个基于 Java 的开源网络爬虫。Nutch 与 Solr 无缝集成，开箱即用。通过 Nutch 采集网页，然后 Solr 实现采集到的网页可被搜索。因此，如果搜索应用需要大规模采集超链接网页，Nutch 可能是一个不错的起点。有关 Nutch 的更多信息，请访问 Nutch 项目主页 <http://nutch.apache.org>。

了解了如何向 Solr 发送文档进行索引，接下来介绍 Solr 如何使用更新处理器处理这些请求。

## 5.6 更新处理器

上一节中使用 HTTP POST 方式向 Solr 发送了新的文档。添加新文档的请求提交给了 Solr 的更新处理器。通常情况下，更新处理器负责索引的所有更新请求，包括提交与优化请求。表 5.7 是更新处理器支持的常见请求类型概览。

表 5.7 更新处理器的常见请求概览

请求类型	说明	XML 举例
添加	向索引添加一个或多个文档，完整举例参见代码清单 5.12	<pre> &lt;add&gt;   &lt;doc&gt;     &lt;field name="id"&gt;1&lt;/field&gt;     ...   &lt;/doc&gt; &lt;/add&gt; </pre>
删除	通过 ID 删除文档，例如，删除 ID 为 1 的文档	<pre> &lt;delete&gt;   &lt;id&gt;1&lt;/id&gt; &lt;/delete&gt; </pre>
按查询删除	删除与 Lucene 查询匹配的文档，例如，删除用户 <code>screen_name=@thelabdude</code> 的所有微博文档	<pre> &lt;delete&gt;   &lt;query&gt;     screen_name:@thelabdude   &lt;/query&gt; &lt;/delete&gt; </pre>

续表

请求类型	说明	XML举例
原子更新	使用乐观锁（Optimistic Locking）更新已有文档的一个或多个字段，参见 5.6.3 节	<pre>&lt;add&gt;   &lt;doc&gt;     &lt;field name="id"&gt;1&lt;/field&gt;     &lt;field update="set" name="       favorites_count"&gt;       12     &lt;/field&gt;   &lt;/doc&gt; &lt;/add&gt;</pre>
提交	向索引提交文档有软提交和硬提交两种选项。在新搜索器打开与预热之前，决定是否等待	<pre>&lt;commit waitSearcher="true"   softCommit="false" /&gt;</pre>
优化	通过合并片段与去掉删除来优化索引	<pre>&lt;optimize   waitSearcher="false"/&gt;</pre>

表 5.7 给出了使用 XML 的更新请求示例，更新请求处理器还支持其他格式，如 JSON、CSV 和 javabin。背后的技术原理是，更新请求处理器根据 HTTP 头部的 Content-Type 来识别请求的格式，例如，Content-Type:text/xml。代码清单 5.16 是 solrconfig.xml 中更新处理器的配置信息。

代码清单 5.16 solrconfig.xml 中更新处理器的配置元素

配置自动提交策略，参见 5.6.1 节。

```
<updateHandler class="solr.DirectUpdateHandler2">
  <updateLog>
    <str name="dir">${solr.ulog.dir:}</str>
  </updateLog>
  <autoCommit>
    <maxTime>15000</maxTime>
    <openSearcher>false</openSearcher>
  </autoCommit>
  <autoSoftCommit>
    <maxTime>1000</maxTime>
  </autoSoftCommit>
  <listener event="postCommit" ...>
    ...
  </listener>
</updateHandler>
```

注册一个更新事件监听器。

配置软自动提交策略，参见 5.6.1 节。

启用事务日志，参见 5.6.2 节。

更新处理器最重要的任务是处理文档提交到索引的请求，并让这些文档可以被搜索。

### 5.6.1 将文档提交到索引

本节介绍 Solr 如何通过将文档提交到索引，让这些文档可被搜索。当一个文档被添加到 Solr 中，在没有提交给索引之前，这个文档无法被搜索。换句话说，从查询的角度看，文档直到提交之后才是可见的。Solr 4 有两种类型的提交：软提交和正常提交（俗称硬提交）。首先介绍正常提交的工作原理，这将有助更好地理解软提交。

#### 正常提交

Solr 的正常提交（硬提交）是将所有未提交的文档写入磁盘，并刷新一个内部搜索器组件，让新提交的文档能够被搜索。搜索器实际上可以看作索引中所有已提交文档的只读视图。参见 4.3 节有关搜索器工作原理的详细讨论。可以这样说，硬提交是花销较大的操作，由于硬提交需要开启一个新搜索器，所以会影响到查询性能。

当正常提交成功后，新提交的文档被安全保存在持久存储器上，不会因为正常的维护操作或服务器崩溃后重启而丢失。出于高可用性考虑，如果磁盘发生故障，就需要一套故障转移方案。第 13 章将讨论高可用性功能。

#### 软提交

软提交是 Solr 4 的一项新功能，支持近实时搜索（Near Real-Time, NRT）。第 13 章会详细讨论近实时搜索。这里将软提交作为文档近乎实时可被搜索到的一种机制，跳过了硬提交的高消耗，例如，刷新到持久存储器就是花销较大的操作。软提交相对而言花销较低，我们可以每一秒都执行一次软提交，使得新近被索引的文档在添加到 Solr 之后很快被搜索到。但要记住，在某一时刻仍然需要执行硬提交操作，以确保文档最终写入到持久存储器上。

综上所述：

- 硬提交让文档可被搜索，由于需要将其写入到持久存储器上，所以花销较大。
- 软提交也可以让文档被搜索，不需要将其写入到持久存储上。

第 13 章将继续讨论这个话题，到时会在 SolrCloud 中讨论近实时搜索。

#### 自动提交

不管是正常提交还是软提交，都可以采用以下三种策略中的任意一种来自动提交文档：

- 在指定时间内提交文档。
- 一旦达到用户指定的未提交文档阈值，就提交那些未提交的文档。
- 每隔特定时间间隔提交所有文档，例如，每隔 10 分钟。

Solr 硬提交与软提交的自动提交行为需要在 `solrconfig.xml` 中进行配置。以下 XML 片段是配置举例, Solr 每隔 10 分钟或最大文档数达到 5 000 个时进行自动提交。

```
<autoCommit>
  <maxTime>600000</maxTime>
  <maxDocs>50000</maxDocs>
  <openSearcher>true</openSearcher>
</autoCommit>
```

每隔 10 分钟提交一次  
(以毫秒为单位)。

每 50 000 个文档  
提交一次。

提交后打开一个新搜索器。

执行自动提交时通常会打开一个新搜索器。Solr 通过指定 `<openSearcher>false</openSearcher>` 可以禁用这一行为。在这种情况下, 该文件将被写入到磁盘, 但在搜索结果中不可见。Solr 之所以提供这个选项, 是为了减少未提交更新的事务日志大小 (参见下一节), 并避免在大规模索引过程中打开太多搜索器。

假设有 500 万个文档需要索引, Solr 的自动提交条件设置为文档数阈值达到 50 000。这意味着, Solr 在索引 500 万个文档过程中会执行 100 次自动提交。如此一来, 索引完所有文档后打开一个新搜索器, 要比打开 100 个新搜索器更现实些。当然, 客户端应用也可以每 100 万个文档发送一次硬提交请求, 这样做的话, 有一些文档很快就可以被搜索了。考虑的重点在于, 是否每次自动提交后都需要打开一个新搜索器。如果索引的文档数量大于自动提交的阈值, 那么可能要考虑设置成 `<openSearcher>false</openSearcher>`。索引完所有文档后, 最后执行一次硬提交。

另外, 不要把表 5.7 中的 `<Commit>` 元素的 `waitSearcher` 属性与 `<autoCommit>` 元素的 `openSearcher` 属性搞混淆了。当发送一个 `<commit>` 请求时, 通常会打开一个新搜索器并进行预热。 `waitSearcher` 属性决定新搜索器完全启动后, 客户端代码是否应该被阻止。正如第 4 章介绍的, 新搜索器的启动需要很长时间, 所以要谨慎使用 `waitSearcher="true"`。

在 `solrconfig.xml` 中使用 `<autoSoftCommit>` 元素也可以自动配置软提交。如果想要设置更小的软提交间隔值, 例如, 每秒 (1000 毫秒), 如下 XML 片段所示:

```
<autoSoftCommit>
  <maxTime>1000</maxTime>
</autoSoftCommit>
```

每隔 100 毫秒软提交一次。

接下来介绍更新处理器的另一个强大功能, 如何确保不会丢失未提交的更新。

## 5.6.2 事务日志

Solr 使用事务日志来确保提交到索引并已接受的更新保存在持久存储器。假设这样的场景, 客户端应用每 10 000 个文档发送一次提交。如果在客户端发送的文档

已被索引，但还没来得及执行提交操作的情况下 Solr 崩溃了。那么，没有事务日志的话，这些未被提交的文档将会丢失。具体来说，事务日志主要有三个作用：

- 支持近实时获取和原子更新。
- 解除提交过程中写入的持久性。
- 通过 SolrCloud 的分片代表支持副本的同步（参见第 13 章）。

在 solrconfig.xml 中一个 Solr 内核的事务日志配置如下：

```
<updateLog>
  <str name="dir">${solr.ulog.dir}</str>
</updateLog>
```

默认位置是 data 目录的子目录 tlog。

每次更新请求都会被记录到事务日志。直到发起提交，事务日志会持续增长。在提交期间会处理活动的事务日志，之后将打开一个新的事务日志。图 5.7 展示了更新请求处理的步骤。

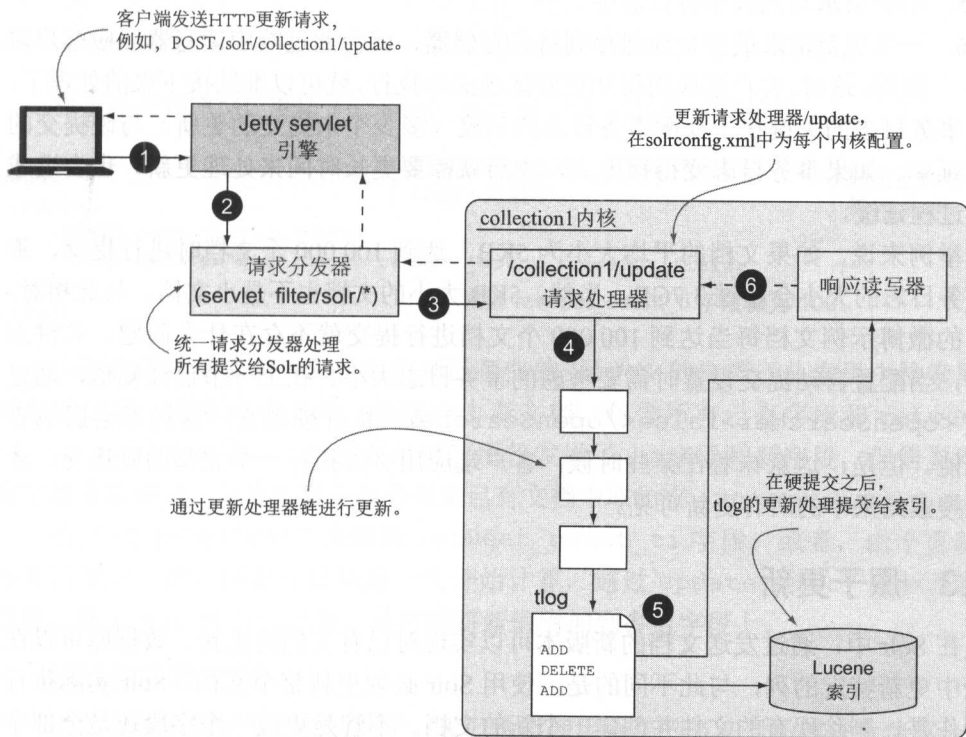


图 5.7 更新请求处理用到的主要组件和事件顺序，例如，添加新的文档

图 5.7 的一些组件，如请求调度器和回复写入器，应该是你熟悉的。这些与第 4 章介绍的查询请求处理组件相同。接下来，通过图 5.5 的事件处理顺序来了解一些重要概念：



1. 客户端应用程序使用 HTTP POST 方式发送一个更新请求，可以是 JSON、XML 或 Solr 内部二进制 javabin 格式。代码清单 5.15 使用 SolrJ 编写了一个示例客户端。
2. Jetty 将此请求发送给 Solr 的 Web 应用程序。
3. Solr 的请求调度器通过请求路径中的 "collection1" 这一部分来确定 Solr 的内核名称。接下来，调度器定位到 /update 请求处理器，它在 solrconfig.xml 的 collection1 内核上注册过。
4. 更新请求处理器对该请求进行处理。当添加或更新文档时，更新处理器依据 schema.xml 依次处理请求中的每个文档的每个字段。此外，该请求处理器将调用一个可配置的更新处理器链，在索引时为每个文档进行额外的处理。第 6 章将举例说明如何在索引期间使用更新请求处理器来检测语种。
5. ADD 请求写入到事务日志中。
6. 一旦更新请求被安全地保存到持久存储器，就会通过响应读写器回应客户端应用。这时，客户端应用得知更新请求成功执行，就可以继续接下来的处理了。

事务日志的关键在于权衡事务日志的长度（多少个未提交的更新）与硬提交的执行频率。如果事务日志变得很庞大，重启就需要更长时间来处理更新，也会造成恢复过程延缓。

举例来说，如果文档的平均大小为 5KB，达到 100 000 个文档时进行提交，那么事务日志的大小会超过 3.7GB。当然，5KB 大小的文档也不是小文档。与此相对，本章的微博示例文档每当达到 100 000 个文档进行提交就不存在什么问题。关键点在于，当配置自动提交设置时需要考虑的事务日志大小。在上一节已经知道，通过设置 `<openSearcher>false</openSearcher>` 执行硬提交，这样不会影响查询性能。但是，这意味着在某些时候，客户端应用必须执行一个完整的硬提交，才能让搜索结果中的所有更新可见。

### 5.6.3 原子更新

在 Solr 中，通过发送文档的新版本可以实现对已有文档的更新。数据库可以在一行中更新特定的列，与此不同的是，使用 Solr 必须更新整个文档。Solr 实际执行的操作是，删除现有的文件并创建一个新的文档。不管是更改一个字段还是全部字段，都会这样执行。

从客户端的角度看，应用程序必须发送整个文档的新版本。对于文档来自于其他来源的应用程序，这不是什么大问题。但是，对于那些将 Solr 作为主要数据存储的应用程序来说，为了更新单个字段而要重新创建全部文档，这可能会产生问题。在实践中，这需要用户查询整个文档，进行更新，并将指定的文档整体发回给 Solr。

对已存在的文档请求所有字段，更新字段的子集，并发送新版本给 Solr。这样的做法在实践中很常见。因此，原子更新是 Solr 的一项新功能，通过它可以实现仅对需要修改的字段进行更新。这让 Solr 更符合数据库的更新操作。Solr 仍可以删除与新建文档，但这些对客户端应用程序代码来说是透明的。

### 字段级别的更新

回到之前的微博搜索应用示例，假设我们要在已有文档上索引一个新字段，用来保存微博的转发次数。我们把这个新字段作为微博热门程度的一个指标。为了简单起见，此字段每天更新一次。当然，我们都知道日统计量是有用的，但为了简单起见，我们把它处理成累积值。此处的重点是掌握原子更新。

将新字段命名为 `retweet_count_ti`，表示这是一个动态字段。因此，添加新字段不需要更新 `schema.xml`。`_ti` 后缀来源于以下 `schema.xml` 中的动态字段定义：

```
<dynamicField name="*_ti" type="tint" indexed="true" stored="true"/>
```

以下是使用 XML 更新 `retweet_count_ti` 字段的示例请求：

```
<add>
  <doc>
    <field name="id">1</field>
    <field update="set" name="retweet_count_ti">100</field>
  </doc>
</add>
```

看起来不太直观，更新必须包含在 `<add>` 元素中。

确定已存在的文档 id 为 1。

设置 `retweet_count_ti` 字段值为 100。

实际执行的操作是，Solr 定位到已存在的 id 为 1 的文档，从索引中检索所有的存储字段，删除已有文档，创建一个新文档，包含所有已有字段和 `retweet_count_ti` 这个新字段。虽然客户端应用仅发送了 id 字段和新字段，但需要对所有字段重新存储。其他所有字段必须从已有文档中取出来。

通过 `update="set"` 来设置 `retweet_count_ti` 字段。或者，由于更新进程每日运行一次，因此可以从前一天开始计算，通过 `update="inc"` 增加已有字段值。除了 `set` 和 `inc` 之外，还可以将新值附加到多值字段上。

### 积极的并发控制

现在考虑稍微复杂一些的例子，我们希望使用众包的方法对微博进行情感分类。简单地说，我们支付给用户一定费用，让他们对每个文档做出正面、中性或负面的分类。情感字段对于找出产品或餐厅负面信息来说是有用的。

完成分类后，每个带着情感标签的微博文档都需要在 Solr 中进行更新。在微博转发示例中，通过自动化处理方式每天更新一次 `retweet_count_ti` 字段。但对

于情感分类, `sentiment_s` 字段的更新可以在任何时间进行。因此, 两个用户在相同时间对同一个文档进行情感标签的更新, 这种情况是可能出现的。当然, 我们也可以通过一些烦琐的过程, 在用户打情感标签之前, 显式锁定文档, 但这会降低处理效率, 完全没有必要。此外, 我们也不希望为一个文档多支付一次分类费用。因此, 需要一些方法来防止同一文档上的并发更新, 也就是积极的并发控制。

为避免冲突, Solr 通过版本跟踪字段 `_version_` 来支持积极的并发控制。在 `schema.xml` 中定义版本字段, 如下所示:

```
<field name="_version_" type="long" indexed="true" stored="true"/>
```

在 Solr 4 中, 不要在 `schema.xml` 中修改或删除这个字段。

当添加一个新文档时, Solr 会自动分配一个唯一版本号。当需要警惕并发更新时, 则在更新请求中包含精确的更新版本。以下是包含具体 `_version_` 的更新请求:

```
<add>
  <doc>
    <field name="id">1</field>
    <field update="set" name="sentiment_s">positive</field>
    <field update="set" name="classified_by_s">SomeUserID</field>
    <field name="_version_">1234567890</field>
  </doc>
</add>
```

根据文档的唯一 ID 来确认要更新的文档。

将 `sentiment_s` 字段设置为“positive”。

记录下分类这条微博用户, 以便计算报酬。

返回此更新基于的文档 `_version_` 字段。

当 Solr 处理此更新时, 它会比较更新请求的 `_version_` 值与文档最新版本。文档的最新版本从索引或事务日志中获得。如果版本匹配, 则 Solr 执行此更新请求; 如果不匹配, 则更新请求失败, 将错误信息返回给用户。客户端应用可以处理错误响应, 让用户知道该文档已被其他用户标注过了。由于假定大多数更新在初次尝试会正常更新, 冲突比较少见, 所以我们称这种方式为“积极的”。

使用 `_version_` 字段执行并发控制会产生一个问题: 客户端应用如何从 Solr 获得当前 `_version_` 值? 最好的方法是使用近实时的 `get` 请求。例如, 在微博搜索示例中, 为了获得 `id` 为 1 的文档的 `_version_` 字段值, 需要发送以下 HTTP `GET` 请求: `http://localhost:8983/solr/collection1/get?id=1&fl=id,_version_`。

不论该文档是否提交到索引, 近实时 `get` 会返回该文档的最新版本, 因此, 近实时 `get` 请求和原子更新依赖于索引开启事务日志。

Solr 还提供 `_version_` 字段的其他并发更新方法。表 5.8 给出了 Solr 根据更新请求的 `_version_` 值采取的更新策略。

表 5.8 Solr 根据 `_version_` 字段值采取的更新策略

若 <code>_version_</code> 字段为	Solr 的行为
>1	版本必须匹配, 否则更新失败
1	文档必须存在
<0	文档不能存在
0	无并发控制需要, 现有值被覆盖

通过这个简单的示例, 我们已经看到了原子更新的强大作用。原子更新是 Solr 数据管理方面的一个新武器。在 Solr 4 中, 你可以通过发送需要更新的字段和待更新文档的唯一标识符, 对已有文档进行更新。

## 5.7 索引管理

由于需要先掌握 Solr 索引创建的知识, 所以第 4 章推迟了有关 `solrconfig.xml` 中索引管理设置的讨论。现在可以介绍 Solr 的索引管理设置了。本节重点介绍最有可能需要调整的索引设置, 首先从索引文档的存储设置谈起。应该说, 大部分 Solr 索引相关的设置仅面向技术专家。这就意味着, 应该谨慎做出修改。默认设置适用于大多数 Solr 的装机情况。

### 5.7.1 索引存储

当文档提交到索引之后, `directory` 目录组件将它们写入到持久存储器。Solr 的目录组件具有以下重要特点:

- 隐藏持久存储的读写细节, 例如, 将文档写入到磁盘或通过 JDBC 在数据库中存储文档。
- 实现特定的存储锁定机制, 防止索引出错。例如, 在操作系统级别上基于文件系统的存储锁定。
- 将 Solr 从 JVM 和操作系统的专有性从解脱出来。
- 启用基础目录方案的扩展机制, 以支持特定应用, 如近实时搜索。

Solr 提供不同的目录方案, 但没有所谓的适用于所有 Solr 装机情况的最佳目录方案。根据 Solr 应用的具体情况, 思考如何确定最佳方案。实践中取决于操作系统本身、JVM 类型及应用场景。第 4 章曾提到, Solr 在启用时就尝试做好开箱即用的配置。首先来看 Solr 索引存储的默认配置, 为进一步修改打下基础。

### 默认存储配置

默认情况下, Solr 为一个内核设置一个数据目录, 将数据存储在本机文件系统中。举例来说, 示例服务器将索引存储在 `$SOLR_INSTALL/example/solr/collection1/data/` 目录。在 `solrconfig.xml` 中使用 `<dataDir>` 元素定位 `data` 目录。

```
<dataDir>${solr.data.dir:</dataDir>
```

← solrconfig.xml 的默认配置; 位置解析到示例服务器的 collection1/data。

`solr.data.dir` 属性是内核的默认数据目录, 但可以在 `solr.xml` 中修改, 例如:

```
<core loadOnStartup="true" instanceDir="collection1/"
  transient="false" name="collection1"
  dataDir="/usr/local/solr-data/collection1"/>
```

← collection1 内核存储数据的目录。

首先要考虑, 索引的数据目录是否有足够的存储容量。此外, 数据目录支持快速读写也很重要, 其中对读性能要多做一些考虑。虽然磁盘 I/O 优化策略超出了本书讲解的范畴, 但有一些基础的知识点需要了解:

- 每个内核不应与其他进程争夺磁盘空间。
- 如果同一台服务器上有多个内核, 建议的做法是每个索引使用独立的物理磁盘进行存储。
- 如果预算允许, 考虑使用高质量、高速磁盘, 如固态硬盘 SSD。
- 花一些时间与系统管理员讨论服务器的 RAID 策略。
- 操作系统中文件系统用于缓存的内存容量也会对磁盘 I/O 需求产生不可小觑的影响。

以上考虑及其对 Solr 性能的影响将在第 12 章中详细讨论。

### 选择一个目录方案

一旦解决了存储方面的顾虑, 还需考虑存储方案的最佳目录方案。Solr 使用 `solr.NRTCachingDirectoryFactory` 启用默认目录, 在 `solrconfig.xml` 中配置 `<directoryFactory>` 元素来指定此目录。

```
<directoryFactory name="DirectoryFactory"
  class="${solr.directoryFactory:solr.NRTCachingDirectoryFactory}"/>
```

`NRTCachingDirectoryFactory` 是 `solr.StandardDirectoryFactory` 的一个封装类, 用来添加对近实时搜索的支持。在运行时, `StandardDirectoryFactory` 根据操作系统与 JVM 类型来选择一个具体的目录方案, 如图 5.8 所示。

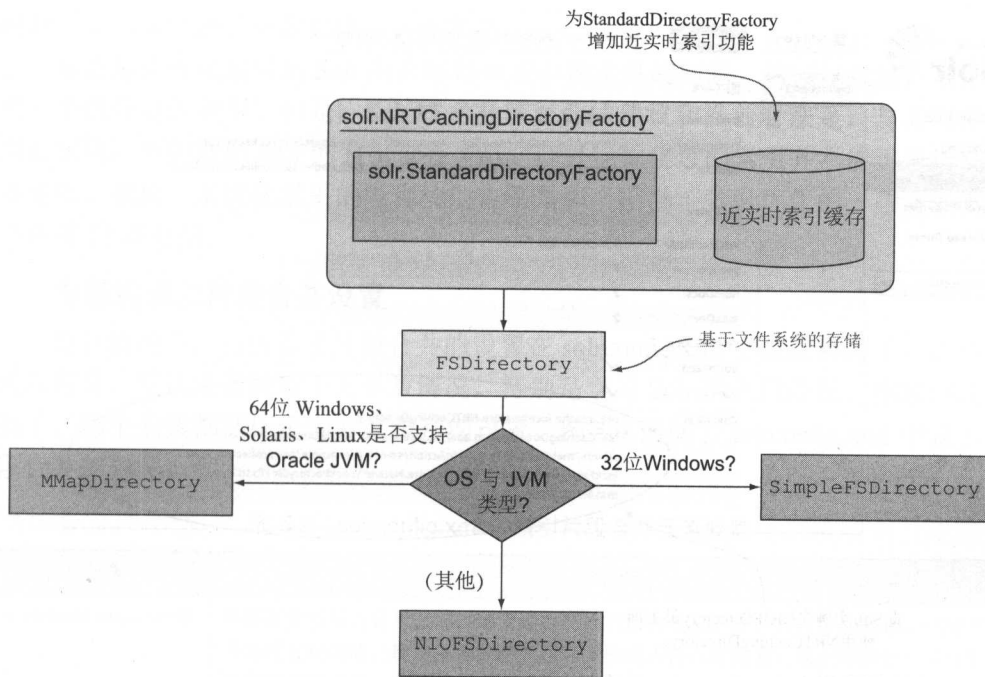


图 5.8 根据操作系统的版本与 JVM 类型，在运行时选择 Solr 目录方案

根据图 5.8，Solr 包含三个基于文件系统的目录备选项。

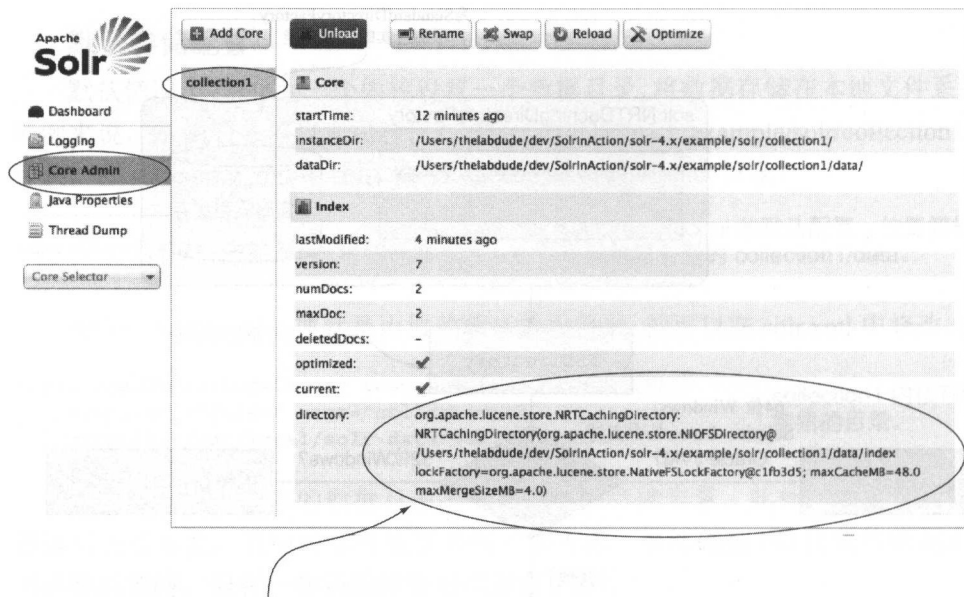
- MMapDirectory：读取索引时使用内存映射 I/O。这是安装了 Oracle JVM 的 64 位 Windows、Solaris 或 Linux 类操作系统的最佳选择。
- SimpleFSDirectory：使用 Java 的 RandomAccessFile 方法。除非是 32 位 Windows 操作系统，否则应避免使用此方法。
- NIOFSDirectory：通过 java.nio 进行优化，以避免同一个文件的同步读操作。这是 JVM 长期存在的一个问题，应避免在 Windows 上使用此方法。

通过 Solr 控制台的内核管理页，为 Solr 服务器指定数据目录。图 5.9 说明了如何寻找示例 collection1 内核的目录信息。

通过在 solrconfig.xml 指定目录来修改默认目录。例如，图 5.9 中 Solr 使用了 NRTCachingDirectory。如果想要修改为 MMapDirectory，则需要对 solrconfig.xml 作如下修改：

```
<directoryFactory name="DirectoryFactory"
  class="${solr.directoryFactory:solr.MMapDirectoryFactory}"/>
```

显式使用  
MMapDirectory 方案。



此Solr实例在NIOFSDirectory最上面  
使用NRTCachingDirectory。

图 5.9 内核管理页中 collection1 内核的活动目录截图

如果运行在 64 位 Windows、Solaris 和 Linux 操作系统上，MMapDirectory 是最佳选择。因为 MMapDirectory 使用了现代操作系统的虚拟内存管理功能，对读性能进行了优化，使其更加智能化<sup>3</sup>。

## 5.7.2 索引片段合并

索引片段是 Lucene 完整索引的子集，具有自包含和只读特点。索引片段写入持久存储器之后，它就无法再更改。向索引添加新文档时，它们被写入一个索引片段中。因此，索引里可能存在许多活动的索引片段。每个查询必须读取了全部索引片段的数据，才能得到完整的搜索结果集。基于这一点考虑，过多的小片段会对查询性能造成负面影响。将多个小索引片段组合成大索引片段的做法称为索引片段合并。

### 索引是否需要优化

优化要求 Lucene 将已有的索引片段合并成一定数量（默认值为 1）的大索引片段。举例来说，由 32 个片段组成的索引在优化之后会变为一个大索引片段。在 Solr 中，优化会对内存、CPU 与磁盘 I/O 的花销较大，特别是对大索引而言。花费数小

<sup>3</sup> 如果想要深入了解 MMapDirectory，推荐查看 Uwe Schindler 的博客文章 *Use Lucene's MMapDirectory on 64bit platforms, please!* (<http://blog.thetaphi.de/2012/07/use-lucenes-mmapdirectory-on-64bit.html>)。



时对一个索引进行全面优化，这很常见。

是否需要优化索引是 Solr 用户邮件列表中最常见的问题。这可以理解，谁不想要一个优化好的索引？但是，Solr 技术社区对此的建议和当前看法是，与其对索引进行优化，不如微调 Solr 的索引合并策略。此外，优化过的索引并不能迅速提高查询速度。相反，未优化索引的查询性能也不是那么糟糕。索引优化的一些有效案例会在第 12 章介绍。

### 专家级索引片段合并设置

默认情况下，有关索引片段合并的设置被注释掉了。这样做的原因是，默认设置适用于大多数情况，特别是学习 Solr 的入门阶段。你应该注意到了，每个元素都被标记为专家级别的设置。图 5.9 列举了 solrconfig.xml 中索引片段合并的相关元素。

表 5.9 solrconfig.xml 中索引片段合并元素概览

元素	用途
ramBufferSizeMB	在缓存文档写入目录之前，用于缓存文档的最大 RAM 值；默认为 100MB。这里不要与提交相混淆。提交是强制所有缓存文件写入持久存储器。增加该值会在内存中缓存更多的文档，在索引期间减少磁盘 I/O
maxBufferedDocs	在写入目录之前，索引创建过程中缓存的文档最大数量；默认是 1 000 个文档。这里不要与提交相混淆。提交时强制所有缓存文档写入持久存储器，并可以进行搜索
mergePolicy	控制 Lucene 如何执行片段合并，例如，确定一次合并的索引片段数量。默认是 TieredMergePolicy，参见第 12 章
mergeFactor	控制一次合并的片段数量；默认为 10。索引的 mergeFactor 最佳值取决于平均文档大小、可用的 RAM 以及所需的索引吞吐量
mergeScheduler	控制片段合并过程；默认设置使用 ConcurrentMergeScheduler 在后台并发执行

这里需要明确，尽管这些专家级设置被注释掉了，但仍然能在后台进行索引片段合并操作<sup>4</sup>。当前的建议是，除非有充分理由修改这些设置，否则跳过优化，直接使用索引片段合并的默认配置即可。当服务器该做索引片段合并时，很可能不作为就是正确的选择<sup>5</sup>。如果搜索应用的索引吞吐量出了问题了，请参照第 12 章修改这些配置。

### 删除处理

当索引片段写入持久存储器之后就不能更改了，这一点现在应该清楚。那么，

4 参考 Mike McCandless 的博客文章 *Visualizing Lucene's segment merges*，了解索引片段合并的可视化 (<http://blog.mikemccandless.com/2011/02/visualizing-lucenes-segment-merges.html>)。

5 更多合并的知识，请参考 *Lucene in Action(2e)* 的 2.13.6 节。

此处的删除并不是从已有的索引片段中删除文档。删除的文档不会从索引中消失，除非包含删除的索引片段被合并。简言之，Lucene 将删除操作记录在一个单独的数据结构中，合并时执行这些删除操作。大多数情况下无须担心合并过程，第 12 章会深入介绍索引片段合并原理及其带来的性能影响。

## 5.8 本章小结

读到这里，你应该对 Solr 的创建索引过程有了很好的理解。回顾一下，首先我们学习了 schema 的设计过程。具体来讲，讨论了文档粒度、文档唯一性，以及字段是否应该被索引、存储或两者兼具的考虑因素。

接下来学习了在 schema.xml 中如何定义字段，包括多值字段与动态字段。多值字段对多字段的文档和多来源的文档与非常有用。另外，为了实现多全字段文本搜索，或在创建索引时对同一文本使用不同文本分析器，我们还学习了 Solr 的 `<copyField>` 用法。

接下来，了解了结构化数据的使用方法。Solr 支持字符串、日期与数值等字段类型。Solr 使用 / 操作符来设置日期值的精确度，例如，小时级别的精度使用 / HOUR。另外，我们还学习了 Solr 基于字典树的字段，用来支持有效的区间查询，以及对数值和日期字段进行排序。通过调整数值字段与日期字段的 `precisionStep` 属性，在较大索引与较慢的区间查询性能之间做出权衡。

在充分熟悉 schema.xml 的基础上，我们学习了如何通过 HTTP 和 SolrJ 向 Solr 发送 XML 和 JSON 文档。除了这两种文档提交方法之外，还介绍了从关系型数据库（DIH）导入文档，使用抽取请求处理器（Solr Cell）对 PDF、微软的 Word 文档等二进制文档进行索引。

当文档处理好之后，需要执行正常提交或近实时搜索的软提交。本章展示了 Solr 如何使用事务日志来避免未提交更新的丢失。除了介绍如何添加新文档，还介绍了如何使用 Solr 的原子更新功能对已有文档进行更新。根据特定的 `_version_` 字段实现乐观的并发控制，以确保正常有序的并发更新。

本章结尾处讨论了 `sorlconfig.xml` 中索引相关的设置。具体来说，介绍了如何使用目录组件对索引进行存储。索引片段合并是避免索引优化的好办法。在没有对索引吞吐量需求有更透彻的理解之前，不要修改索引片段合并的设置。第 12 章会深入讨论专家级的索引性能影响因素。

下一章将深入介绍索引过程中的文本分析。读完第 6 章，你就应该有能力为搜索应用设计与实现一个强大的索引解决方案了。

# 文本分析

## 本章要点

- 使用 Solr 分析表单进行测试
- 为高级文本分析自定义字段类型
- 通过 Solr 的插件框架扩展文本分析

我们在第 5 章已经了解了 Solr 索引的工作原理，并知道如何在 `schema.xml` 中定义非文本字段。本章将通过文本分析进一步了解索引过程。

文本分析消除了索引词项与用户搜索词项之间的语言差异，让用户在搜索 `buying a new house` 时能找到 `purchasing a new home` 这样的文档。本章介绍如何通过配置 Solr，让包含 `house` 的查询与包含 `home` 的文档之间建立匹配关系。

如果配置恰当，文本分析就能允许用户使用自然语言进行搜索，而无须考虑搜索词项的所有可能形式。谁也不想看到用户构造出如下的查询表达式：`buying house OR purchase home OR buying a home OR purchasing a house ...`

用户可以使用自然语言来搜索他们需要的信息，这是提供良好搜索体验的基础。鉴于 Google 和其他搜索引擎的广泛使用，用户往往会期望搜索引擎非常智能，而搜索的智能化就是从优秀的文本分析开始的！

文本分析目前不仅用于消除词项之间的表面差异，还用来解决更复杂的问题，例如，特定语种解析、部分词性标注与词形还原等。如果你不熟悉这些专业术语，

不用担心，接下来会详细讲解。Solr 包含一个文本分析可扩展框架，可以移除常见词汇（也就是停用词），以及执行其他更复杂的文本分析任务。Solr 文本分析框架提供了强大的功能与灵活的扩展性，但对初学者而言，这个框架过于复杂，令人望而却步。我们常说事物总有两面性，Solr 有可能解决非常复杂的文本分析问题，但使得原本一些简单的分析任务做起来反倒变麻烦了，杀鸡焉用牛刀的感觉。Solr 在示例 `schema.xml` 文件中预置了许多字段类型，这样做的原因是，确保初学者在接触开箱即用的 Solr 时，能较容易地开始使用文本分析。学完本章，相信你有能力利用这个强大的框架来分析绝大多数内容。

本章主要介绍 Solr 的文本分析方法，帮助你思考如何为你的文档设计文本分析解决方案。为达到此目的，我们要解决一个稍微复杂的文本分析问题，以证明文本分析机制与策略是有效的。具体来说，本章将介绍 Solr 文本分析的一些基本组件：

- Solr 文本分析的基本元素，包括分析器、分词器与分词过滤链。
- 当索引创建与查询处理时，如何在 `schema.xml` 中为文本分析自定义字段类型。
- 常见的文本分析策略，例如，移除停用词、小写转换、移除重音、同义词扩展及词干提取等。

掌握了文本分析基础模块之后，我们将解决一个难度稍大的文本分析问题，借此熟悉 Solr 文本分析的一些高级功能。具体来说，我们将学习如何对类似 Twitter 的微博内容进行分析。微博带来了与以往不同的挑战，用户如何使用微博搜索，这些需要我们认真思考。微博内容的文本分析任务如下：

- 将词项中重复字符的数量减少到两个，如 `yummm`。
- 保留主题标签符号 `#` 与提及符号 `@`。
- 使用自定义分词过滤器处理诸如 `bit.ly` 这样的短网址。

## 6.1 微博文本分析

这里以第 5 章的微博搜索应用为例。简而言之，我们要设计与实现一个大众社交媒体网站（如 Twitter）的微博内容搜索解决方案。因为本章聚焦文本分析，所以需要进一步了解微博示例文档的文本字段。以下是打算分析的文本：

```
#Yummm :) Drinking a latte at Caff  Grecco in SF's historic North Beach...  
Learning text analysis with #SolrInAction by @ManningBooks on my i-Pad
```

正如本章开头所介绍的，文本分析的主要目标是让用户使用自然语言进行搜索，无须考虑搜索词项的所有可能表达形式。图 6.1 中用户通过文本字段搜索 `San Francisco north beach coffee`，这是一个自然语言查询，用户期望找到北

边海滩上所有不错的咖啡屋。也许我们的用户正在通过搜索社交媒体内容，寻找在北边海滩喝咖啡的好地方。

The screenshot shows a web browser window titled "Microblog Search Tool" with the URL "http://example.com/SolrInAction/TextAnalysis". The search form includes the following fields:

- By User:** A text input field containing "@thelabdude".
- Type:** A dropdown menu with options "Post", "Reply", and "Retweet".
- Language:** A dropdown menu with "English" selected.
- Date Range:** A dropdown menu with "After" selected, followed by a date input field containing "05/01/2012" and a calendar icon.
- With Text:** A text input field containing "San Francisco north beach coffee".
- Search:** A button to execute the search.

Below the search form, the **Search Results** section displays a single result:

@thelabdude on May 22, 2012 @ 09:30 AM ♥ Favorited by 10 users  
 #Yummm :) Drinking a latte at Caffé Grecco in SF's historic North Beach... Learning text analysis with #SolrInAction by @ManningBooks on my iPad

Annotations on the right side of the image point to specific elements:

- Search表单让用户可以对所有已定义的微博文档字段进行搜索。
- 用户名
- 类型
- 语种
- 时间戳
- 微博正文
- 当用户搜索时，由于文本分析的作用，以下示例微博被视为相关的匹配结果。

图 6.1 第 5 章微博搜索应用的表单查询界面

我们判断，上面这条微博应该是这个查询的优先匹配结果。虽然从精确的文本匹配来看，San Francisco、north beach 与 coffee 这三个词并没有出现在这条微博里。当然，返回的文档里包含了 North Beach。那么问题来了，只有指定搜索的索引不区分大小写，才能实现匹配。由此判断，因为用户搜索使用的词项与文档中的词项存在关系，如表 6.1 所示，所以这条微博文档应该是这个查询的优先匹配结果。

表 6.1 查询文本与示例推文文本的匹配

用户搜索的文本	文档的文本
San Francisco	SF's
north beach	North Beach
Coffee	latte, Caffé

因此，摆在我们面前的任务是，使用 Solr 文本分析框架将微博文本转换为一种

易于寻找的形式。图 6.2 展示了 Solr 文本分析框架对微博文本的转换情况，目前你可以把这个框架想象成一个黑盒。本章其余部分将打开这个黑盒，介绍其工作原理。

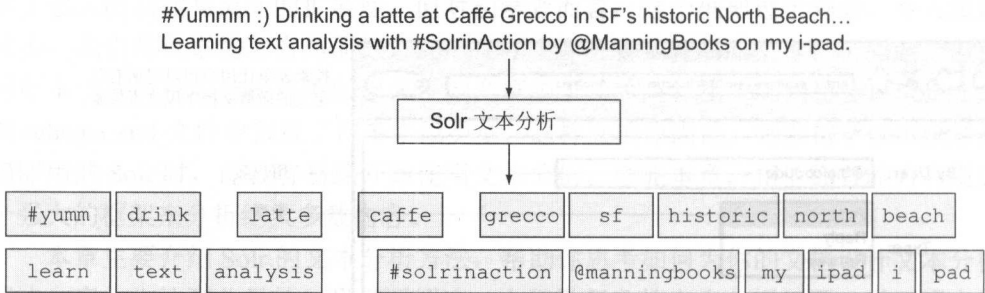


图 6.2 将微博文本转换成更适合搜索的形式。每个框表示 Solr 索引中的唯一词项，这些词项是对这条微博进行文本分析的结果。词项之间的空格表示被索引排除在外的停用词

你能发现使用了哪些转换吗？表 6.2 提供了 Solr 文本分析的各种工具的核心转换一览表，后面会详细介绍。注意到，单独看每个转换似乎都相当简单，但它们合在一起就会大大改善搜索体验。

表 6.2 微博文本使用 Solr 文本分析工具转换一览表。注意，所有的转换都是通过 Solr 内置的文本分析工具来实现的，不需要自己编写代码！

原始文本	转换	Solr 的做法
所有词项	Lowercased (SF's → sf's)	LowerCaseFilterFactory
a, at, in, with, by, on	从文本中移除	非常常见的词项称为“停用词”，使用 StopFilterFactory 将它们从文本中移除
Drinking, learning	drink, learn	使用 KStemFilterFactory 提取词干
SF's	sf, san francisco	使用 WordDelimiterFilterFactory 移除撇号 ('s)
Caffé	caffe	使用将变音符 éASCIIFoldingFilterFactory 转换为 e
i-Pad	ipad, i pad	使用 WordDelimiterFilterFactory 恰当地处理连字符
#Yummm	#yummm	重复的字母使用 PatternReplaceCharFilterFactory 缩小到最多重复两次
#SolrinAction, @ManningBooks	#solrinaction, @manningbooks	#与@使用 WhitespaceTokenizer 与 WordDelimiterFilterFactory 原样保留

有些 Solr 类名看起来吓人，不用担心，本章会依次介绍每个分析工具。现在，我们使用 Solr 内置工具对这条微博进行一些有趣的转换。

我们使用 Solr 的 ASCIIIFoldingFilterFactory 类把 caffè 转换成 caffe，



这意味着, 用户搜索时不需要输入变音符 é。如果没有这个转换, 用户搜索 `caffe` 时找不到索引为 `caffé` 的文档。`i-Pad` 中间有一个连字符, 我们使用 `WordDelimiterFilterFactory` 将其转换成词项 `ipad` 和 `i pad`, 添加到索引里。这意味着, 包含 `ipad`、`i pad` 或者 `i-pad` 的查询都会匹配出结果。`iPad` 其实才是正确的拼写方式, 但在搜索里要尽可能容纳词项的各种简单变化。

`Solr` 的 `PatternReplaceCharFilterFactory` 类使用正则表达式实现字符和词项的替换。例如, 在社交媒体内容 (如微博) 中经常出现重复字母的单词, 这样做是为了表达情感 (例如 `yummm`)。从搜索角度来看, `yummm` 和 `yumm` 基本上是一样的。因此, 我们将重复字母最多保留 2 个, 这样可以减少索引中唯一词项的数量。本章随后会介绍如何在 `Solr` 中使用正则表达式实现这个变换。

值得一提的是, 之前介绍的转换都是由 `Solr` 内置工具实现的。也就是说, 我们只需在 `Solr` 的 `schema.xml` 中进行配置, 无须编写任何 Java 代码。虽然 `Solr` 的内置工具很强大, 但有时你还是需要对它进行功能扩展。6.4.3 节会介绍如何使用 `Solr` 的插件框架来处理社交媒体内容中诸如 `bit.ly` 这样的短网址。

此时, 你应该对 `Solr` 的文本分析印象还不错, 想要知道如何上手。`Solr` 提供了丰富且强大的文本转换工具, 如何对索引过程中的文档应用这些工具呢? 下一节我们将从字段类型着手进行文本分析。

## 6.2 基础文本分析

第 5 章中讲过, `schema.xml` 的 `<types>` 部分使用 `<fieldType>` 元素定义了文档中所有可能的字段, 每个 `<fieldType>` 定义了字段的格式, 以及在索引与查询中该字段如何进行分析。`Solr` 的示例 `schema` 提供了丰富且可扩展的字段类型定义列表, 能够满足大多数搜索应用的需要。如果 `Solr` 的预定义字段类型无法满足需要, 可以使用 `Solr` 插件框架创建自己的字段类型。本章最后一节会介绍一个 `Solr` 插件框架的例子。

如果所有字段都是结构化数据, 如语种代码和时间戳, 那就没必要使用 `Solr`。关系型数据库更适合对结构化数据进行索引与搜索。非结构化文本的处理才是 `Solr` 擅长的。因此, `Solr` 的示例 `schema` 预定义了很多有用的文本分析字段类型。代码清单 6.1 是 `text_general` 字段的 XML 定义, 这是一种较为简单的字段类型, 我们将它作为微博文本分析的起点。



代码清单 6.1 普通文本分析的字段类型示例

```

<fieldType name="text_general"
  class="solr.TextField"
  positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
      words="lang/stopwords_en.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory"
      ignoreCase="true" words="lang/stopwords_en.txt"/>
    <filter class="solr.SynonymFilterFactory"
      synonyms="synonyms.txt"
      ignoreCase="true" expand="true"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>

```

根据数据的类型使用简短、描述性的名称。

为索引文档定义分词器。

为分析查询表达式定义分词器。

分词器将字段文本分解为词元。

本章的例子会对 Solr 示例自带的 schema.xml 做略微改动。建议将 Solr 示例自带的 schema.xml 替换成 \$SOLR\_IN\_ACTION/example-docs/ch6/schema.xml 这个定制版。具体来说，你需要使用以下命令覆盖 \$SOLR\_INSTALL/example/solr/collection1/conf/schema.xml：

```

cp $SOLR_IN_ACTION/example-docs/ch6/schema.xml
➡ $SOLR_INSTALL/example/solr/collection1/conf/

```

此外，还需要将 wdfftypes.txt 文件复制到 conf 目录下：

```

cp $SOLR_IN_ACTION/example-docs/ch6/wdfftypes.txt
➡ $SOLR_INSTALL/example/solr/collection1/conf/

```

由于前面章节已经索引了一些测试文档，为避免干扰，我们删除 data 目录下的所有文件，从一个空的搜索索引开始。

```

rm -rf $SOLR_INSTALL/example/solr/collection1/data/*

```

复制好定制的 schema.xml 和 wdfftypes.txt 之后，在 Solr 管理控制台的内核管理页上重载 collection1 内核，或重启 Solr 服务器。

在代码清单 6.1 中, 为便于理解, 我们将这个 XML 定义分解成几部分。在顶部, 定义了一个 `<fieldType>`。处理文本数据的字段需要指定其类属性为 `solr.TextField`。Solr 会对这些文本进行分析。此外, 还需要给该文本字段命名, 名称应体现待分析文本的特点。例如, 当不清楚分析的文本的语种时, 命名为 `text_general` 是一个不错的选择。

## 6.2.1 分析器

在 `<fieldType>` 元素中至少定义一个 `<analyzer>`, 以确定如何分析文本。常见的做法是定义两个单独的 `<analyzer>` 元素: 一个用于分析索引时的文本, 一个用于分析用户搜索时输入的文本。`text_general` 字段类型就使用了这种方式。稍微思考一下, 为什么要对索引与查询使用不同的分析器? 对索引文档与查询处理进行文本分析, 后者往往需要进行更多的分析。例如, 查询的文本分析中通常会添加同义词, 而索引的文本分析不会这样做, 同义词会增大索引体积, 这样处理对同义词管理也相对容易些。随后会对这种方式举例说明。

虽然可以定义两个单独的分析器, 但查询词项的分析器必须兼容索引时的文本分析方式。考虑这样一种情况: 分析器只对索引词项进行小写转换, 不对查询词项进行小写转换。那么, 由于索引中只有 `north` 和 `beach`, 用户搜索 `North Beach` 就无法找到前面提到的那条微博。

## 6.2.2 分词器

在 Solr 中, 每个 `<analyzer>` 将文本分析过程分为两个阶段: 语词切分 (解析) 和语词过滤。严格来说, 还有第三阶段, 即语词切分之前的预处理阶段, 这个阶段可以使用字符过滤器。6.3.1 节会详细介绍字符过滤, 这里重点介绍语词切分和语词过滤器。

在语词切分阶段, 文本会以各种解析形式被拆分成词元流。`WhitespaceTokenizer` 是最基础的分词器, 仅适用空格拆分文本。`StandardTokenizer` 则更为常见, 它使用空格和标点符号拆分出词项, 而且可用于处理网址、电子邮件地址和缩写词。分词器的定义需要指定 Java 实现的工厂类。要使用常见的 `StandardTokenizer`, 需要指定分词器的类为 `solr.StandardTokenizerFactory`。

```
<tokenizer class="solr.StandardTokenizerFactory"/>
```

在 Solr 中，因为大多数分词器需要提供参数构造器，所以必须指定为工厂类，而不是底层的分词器实现类。通过使用工厂方法，Solr 提供了在 XML 中定义分词器的标准做法。在后台，每个工厂类知道如何将 XML 配置属性“翻译”为构造特定分词器实现类的一个实例。所有分词器会生成词元流，可以使用过滤器进行处理，执行词元的某种转换。

### 6.2.3 分词过滤器

分词过滤器对词元执行以下三种操作中的一种。

- 词元转换：改变词元的形式，例如，所有字母小写或词干提取。
- 词元注入：向词元流中添加一个词元，例如，同义词过滤器的做法。
- 词元移除：删除不需要的词元，例如，停用词过滤器的做法。

过滤器可以链接在一起，对词元进行一系列的转换处理。过滤器的顺序很重要，你不希望看到，应该在小写转换过滤器之后使用的过滤器提前被使用的情况出现。

让我们从 StandardTokenizer 开始来了解 Solr 对示例微博的整个分析过程。

### 6.2.4 StandardTokenizer

此时，你应该已经熟悉了 schema 的设计过程，以及在 schema.xml 中如何定义字段和字段类型。让我们运用这些知识对第 5 章的示例微博进行基础的文本分析。文本分析的第一步是，确定如何使用分词器将文本解析为词元流。从 StandardTokenizer 的使用开始，这个分词器是许多 Solr 和 Lucene 项目的首选解决方案，这个分词器使用空格和标点符号来拆分文本，这点做得很好，还能轻松地处理首字母缩写词和缩略词。我们通过示例微博的解析来看看这个分词器的作用：

```
#Yummm :) Drinking a latte at Caffé Grecco in SF's historic North Beach...  
Learning text analysis with #SolrInAction by @ManningBooks on my i-Pad
```

如果熟悉用户创建内容（User-Generated Content）的社交网络（如 Twitter），你会发现，与大多数微博相比，这条微博的语法结构没什么问题，但从文本解析的角度看，这条微博在处理上会有一些有趣的挑战。图 6.3 是 StandardTokenizer 生成的词元。

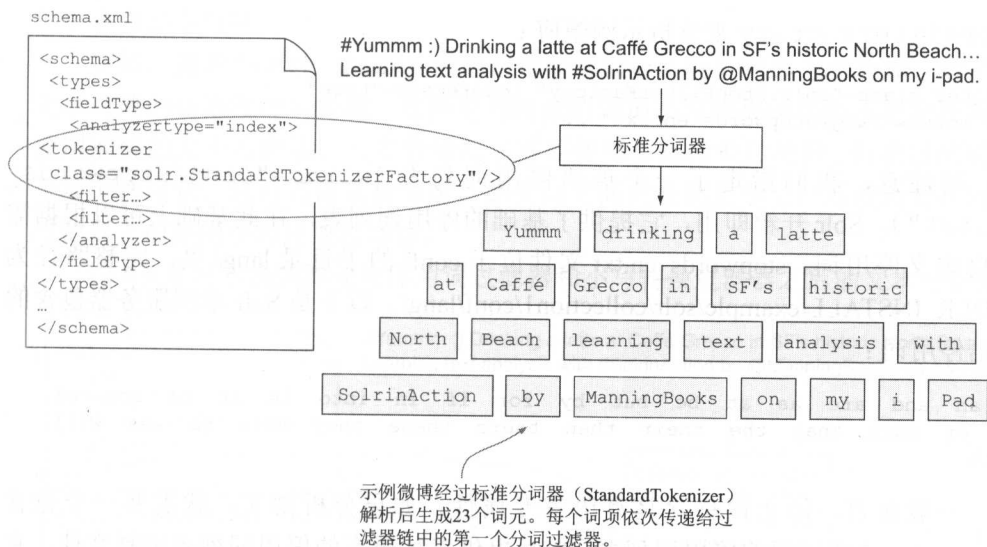


图 6.3 StandardTokenizer 对示例微博进行分析，得到 23 个词元

如你所见，StandardTokenizer 成功地将这条微博拆分成由 23 个不同词元组成的词元流。具体来说，StandardTokenizer 提供以下功能：

- 通过空格、标准的标点符号（如句号、逗号、分号等）拆分文本。注意，省略号和表情符号 :) 被移除了。
- 保留互联网域名和电子邮件地址，分别作为单个词元。
- 连字符词项拆分成两个词元，例如，i-Pad 被拆分为 i 和 Pad。
- 支持可配置的最大词元长度属性，默认值是 255。
- 从主题标签和提及符号中去除 # 号和 @ 号。

接下来，我们来看 Solr 提供的用于基础文本分析的几种常见分词过滤器。还是以那条微博为例，在文本添加到索引之前，还要解决词元流相关的一些问题。首先，一些极为常见的词项，例如，a 和 in 仅是语法要求，它们对区分文档几乎没有任何价值。大多数索引文档中出现的常见词被称为停用词，使用 StopFilterFactory 可以轻松地从词元流中移除它们。

### 6.2.5 使用 StopFilterFactory 移除停用词

Solr 进行文本分析时会使用 StopFilterFactory 移除词元流中的停用词，这些停用词对用户查找相关文档几乎没有价值。在索引时移除停用词会有效减少索引大小，提高搜索性能。这样做减少了 Solr 将要处理的文档数据数量，也减少了对包含停用词的查询进行相关度计算时的词项数量。代码清单 6.1 定义了

StopFilterFactory 来分析示例微博：

```
<filter class="solr.StopFilterFactory" ignoreCase="true"
  words="lang/stopwords_en.txt" />
```

请注意，我们指定了一个英语停用词列表（words="lang/stopwords\_en.txt"）。Solr 开箱即用，它提供了基础的停用词列表，在此基础上可以根据需求自定义停用词。stopwords\_en.txt 文件位于 conf/ 的子目录 lang/ 里，完整路径为 \$SOLR\_INSTALL/example/solr/collection1/conf/lang/。以下是 Solr 示例服务器包含的英语停用词：

```
a an and are as at be but by for if in into is it no not of
on or such that the their then there these they this to was will
with
```

一般而言，停止词移除具有语言专属性。如果分析德文，就需要一个包含 die、ein 这样词项的停用词列表。Solr 提供多种语言的停用词列表定制文件，它们位于 Solr 各个内核 conf/ 的子目录 lang/ 里。

### 停用词移除的高级方法

谷歌拥有一项停用词处理专利，在索引时处理所有停用词，通过判断检索到的文档集合中是否使用停用词，选择性地移除查询中的停用词，请参阅 <http://www.google.com/patents/US7945579>。谷歌的停用词专利方法是搜索提供者如何使用高级文本分析来提升行业竞争力的一个有力做法。即使是停用词移除这样简单的东西，也不存在一个全能的解决方案。

## 6.2.6 使用 LowerCaseFilterFactory 对词项进行小写转换

LowerCaseFilterFactory 将词元的所有字母转换成小写。对 6.2.4 节的示例微博进行索引时，ManningBooks 会被转换为 manningbooks。这样一来，包含 MANNINGBOOKS、ManningBooks、manningbooks 以及其他大小写混合形式的用户查询都可以找到这条微博。定义此过滤器很简单：

```
<filter class="solr.LowerCaseFilterFactory"/>
```

与停用词的情况类似，是否需要对所有词项使用小写转换过滤器有时并不好判断。举例来说，一句话中间的单词首字母大写通常表示一个专有名词。在示例微博中，如果将用户寻找的 North Beach 作为专有名词进行识别，搜索结果的准确度就会提升。由于 north 和 beach 都是各种语境中常用的普通单词，首字母大写的词可

以提升查询专指度。

例如, 搜索 North Beach 的用户可能对旧金山附近的地方感兴趣, 对某岛屿北部海滩的高浪没什么兴趣。这种更为细致的词项小写转换做法要求用户在搜索时使用准确的大小写拼法。大多数情况下需要使用小写转换过滤器, 但在过滤链的哪个环节使用它, 这点还需认真考虑。如果同义词列表都是小写, 那就需要在同义词过滤器之前使用小写转换过滤器。

图 6.4 是对示例微博使用停用词过滤器和小写转换过滤器之后的文本分析结果。

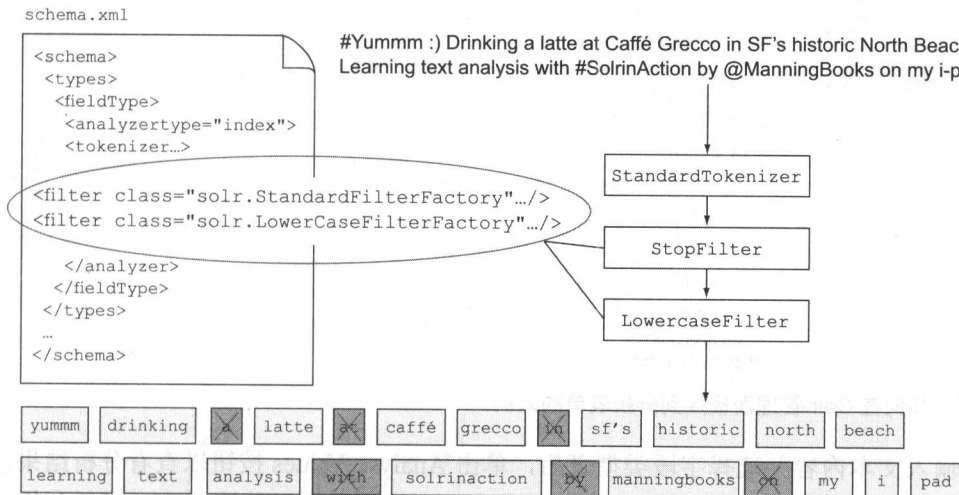


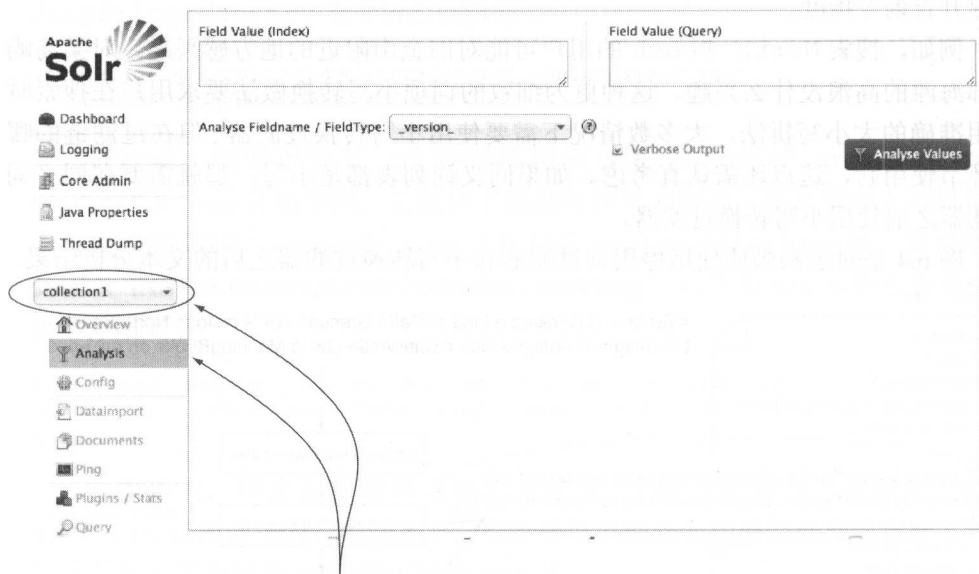
图 6.4 使用 StandardTokenizer、StopFilter 和 LowercaseFilter 进行拆分之后的待索引文本。打 X 的词项是停用词, 不包含在索引中

至此, 我们学习了如何对示例微博进行基础的文本分析。接下来, 我们用学到的文本分析知识进行实际操作。

### 6.2.7 通过 Solr 分析表单进行文本分析测试

Solr 提供了一个简单文本分析表单, 允许用户对示例文本进行分析测试, 无须构建索引文档。在没有索引文档的情况下, 此表单还支持示例文档的查询匹配测试。确定 Solr 服务器在运行, 在浏览器中键入 <http://localhost:8983/solr/>, 进入管理控制台, 点击 collection1, 如图 6.5 所示。

当浏览器载入表单后, 在 Field Value(Index) 文本框中输入示例微博内容, 在下拉列表中选择字段类型为 text\_general, 如图 6.6 所示。



在Solr管理面板，  
点击collection1，  
找到Analysis面板入口。

图 6.5 如何在 Solr 管理面板找到分析表单的入口

输入文本内容并选择字段类型之后，单击 **Analyse Values** 按钮以查看分析结果。在分析测试表单之下，Solr 使用 `text_general` 字段类型对待索引文本进行分析，给出了该文本内容的分析步骤。文本解析首先使用 `StandardTokenizer (ST)` 进行分词，然后依次使用 `StopFilter (SF)` 和 `LowercaseFilter (LCF)` 进行过滤。

接下来，执行一个查询，看看是否能与示例微博实现匹配。在 **Field Value(Query)** 文本框中输入 `drinking a latte`，如图 6.7 所示。单击 **Analyse Values** 按钮，Solr 会高亮显示在文档与查询中同时出现的词项，这里是 `drinking` 和 `latte`。

接下来，输入 `San Francisco drink cafe ipad`，再次单击 **Analyse Values** 按钮。这是一个无意义的查询，这里仅用于说明示例文档匹配的情况。这个无意义的查询足以说明，词项之间微小的差异可能会导致用户错失高度相关的文档。这里没有一个查询词项能匹配出这个示例文档！我们可以很容易地看出查询词项与文档词项的相似程度，但对 Solr 而言，它们之间没有关系！我们需要更好的文本分析方法来解决不匹配的问题。

在第一轮中，我们做了很少的工作就解决了文本解析与基础分析的很多问题。不过，仍有一些问题有待解决，这些问题阻碍了用户顺利找到这条示例微博。你能看出这条微博文本中还存在哪些潜在问题吗？将你的思考与图 6.8 进行对比。





图 6.6 在 Solr 分析表单中使用 text\_general 字段的分析处理步骤

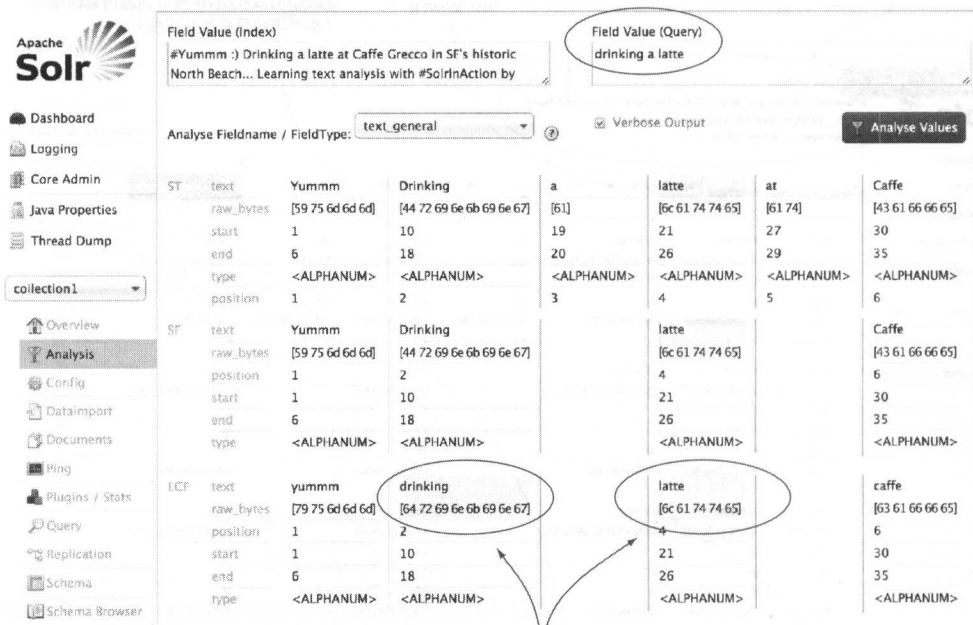
除非对微博和其他社交媒体内容内容进行索引，否则你可能不会遇到图 6.8 中提到的分析问题。一般来说，重点是研究有代表性的索引文档，以此确定需要分析的类型，就像这里的做法。

很明显，text\_general 字段类型提供的基础文本分析已经无法满足当前的需求。因此，我们需要自定义一个新的字段类型，利用之前讨论过的工具以及一或两个新工具来具体实现。

## 6.3 为微博文本自定义一个字段类型

我们已经在社交媒体文本分析的道路上取得了一定进展，但还有一些明显的问题需要解决。本节通过介绍其他的 Solr 内置文本分析工具来解决这些问题。由于 Solr 预定义的字段类型无法满足我们的所有需求，因此我们在 schema.xml 中自定义

一个新字段类型。在 schema.xml 的 <types> 元素下增加 text\_microblog 字段，如代码清单 6.2 所示。



匹配的词语

图 6.7 在 Solr 的分析面板上搜索 drinking a latte，与之匹配的文档中会高亮显示 drinking 和 latte 这两个词项

重复字母在社交类文本中很常见。我们可能需要设置重复字母出现次数为2。

移除变音符，以便搜索caffè时，可以找到包含caffé的文档。

当搜索SF或San Francisco时，s前面的撇号会影响搜索。

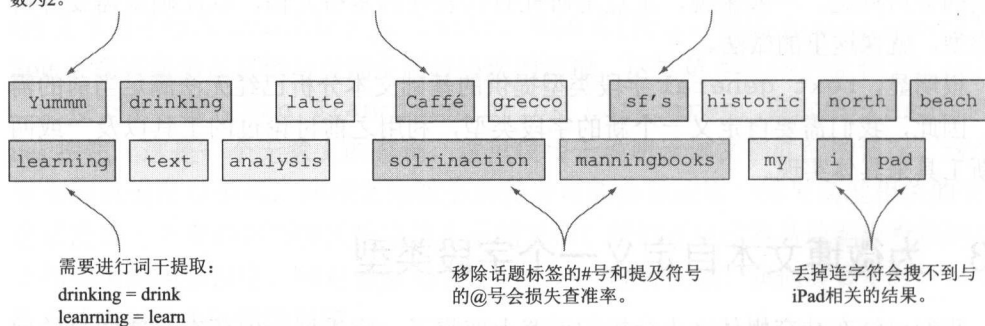


图 6.8 使用小写转换过滤器和停用词过滤器之后，示例微博文本分析仍存在的遗留问题

代码清单 6.2 为微博文本自定义字段类型

```

<types>
...
  <fieldType name="text_microblog" class="solr.TextField"
    positionIncrementGap="100">
    <analyzer type="index">
      <charFilter class="solr.PatternReplaceCharFilterFactory"
        pattern="([a-zA-Z])\1+"
        replacement="$1$1"/>
      <tokenizer class="solr.WhitespaceTokenizerFactory"/>
      <filter class="solr.WordDelimiterFilterFactory"
        generateWordParts="1"
        splitOnCaseChange="0"
        splitOnNumerics="0"
        stemEnglishPossessive="1"
        preserveOriginal="0"
        catenateWords="1"
        generateNumberParts="1"
        catenateNumbers="0"
        catenateAll="0"
        types="wdfftypes.txt"/>
      <filter class="solr.StopFilterFactory"
        ignoreCase="true"
        words="lang/stopwords_en.txt"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.ASCIIFoldingFilterFactory"/>
      <filter class="solr.KStemFilterFactory"/>
    </analyzer>
    <analyzer type="query">
      <charFilter class="solr.PatternReplaceCharFilterFactory"
        pattern="([a-zA-Z])\1+"
        replacement="$1$1"/>
      <tokenizer class="solr.WhitespaceTokenizerFactory"/>
      <filter class="solr.WordDelimiterFilterFactory"
        splitOnCaseChange="0"
        splitOnNumerics="0"
        stemEnglishPossessive="1"
        preserveOriginal="0"
        generateWordParts="1"
        catenateWords="1"
        generateNumberParts="0"
        catenateNumbers="0"
        catenateAll="0"
        types="wdfftypes.txt"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.ASCIIFoldingFilterFactory"/>
      <filter class="solr.StopFilterFactory"
        ignoreCase="true"
        words="lang/stopwords_en.txt"/>
      <filter class="solr.KStemFilterFactory"/>
    </analyzer>
  </fieldType>
</types>

```

将新的字段类型添加在  
<types> 中已有的字段  
类型后面。

索引文档  
的分析器。

处理用户查询  
的分析器。

```

<filter class="solr.SynonymFilterFactory"
  synonyms="synonyms.txt"
  ignoreCase="true"
  expand="true"/>
</analyzer>
</fieldType>
</types>

```

同义词处理在查询端执行，索引阶段不处理同义词。

你应该可以读懂这个字段类型定义的结构及其包含的一些元素，其中还有一些内容我们还没有讨论。表 6.3 是本节介绍的新工具一览表。

表 6.3 微博文本分析需要用到的其他 Solr 文本分析工具清单

Solr 工具	说明
PatternReplaceCharFilterFactory	在分词之前使用正则表达式替换字符
WhitespaceTokenizerFactory	仅以空格分隔文本
WordDelimiterFilterFactory	智能地分隔标点符号及大小写，处理特殊字符，如话题标签的 # 号与提及符号的 @ 号
ASCIIFoldingFilterFactory	如有可能，将变音符转换为等价的 ASCII 值
KStemFilterFactory	提取英文文本的词干，比起波特词干器（Porter Stemmer），它不那么激进
SynonymFilterFactory	在查询中加入常见词的同义词

后面会依次介绍这些工具。首先，使用正则表达式来移除词项中的重复字母，如 yummm。

### 6.3.1 使用 PatternReplaceCharFilterFactory 折叠重复的字母

在 Solr 中，传入的字符流在分词处理之前，字符过滤器对其进行预处理。与分词过滤器类似，CharFilters 作为过滤链中的一环，可以对文本字符进行添加、更改和移除。Solr 4 内置了三个 CharFilters，具体如下。

- solr.MappingCharFilterFactory：使用外部配置文件中定义的字符进行替换。
- solr.PatternReplaceCharFilterFactory：使用正则表达式进行字符替换。
- solr.HTMLStripCharFilterFactory：从文本中去除 HTML 标记。

跟 Solr 的大多数功能一样，你可以使用插件框架设计自己的过滤器。在这三个过滤器中，PatternReplaceCharFilterFactory 似乎最适合当前的文本分析需求。微博内容一般不包含嵌入的 HTML，这里也不需要映射任何字符。因此，这里不展开介绍 MappingCharFilterFactory 和 HTMLStripCharFilterFactory，有关这两个过滤器的详细内容请参阅 Solr wiki (<http://wiki.apache.org/solr/>)。让我们看看如何使用 PatternReplaceCharFilterFactory 来解决示例微博分

析的一些问题。

`solr.PatternReplaceCharFilterFactory` 使用正则表达式来过滤字符。要配置这个 Java 工厂方法，需要定义两个属性：`pattern` 和 `replacement`。`pattern` 是一个正则表达式，用来识别文本中想要替换的字符。`attribute` 指定用什么值去替换匹配到的字符<sup>6</sup>。

如果你不熟悉正则表达式，不用担心，在大多数情况下都可以使用 Google 这些搜索引擎从网上找到需要的正则表达式。

我们使用 `<charFilter>` 解决这个麻烦的重复字母问题，如 `yummm`。要将重复的字母减少到最多出现 2 个，我们需要使用正则表达式识别出重复字母序列，相应的正则表达式为 `([a-zA-Z])\1+`。解释一下，`[a-zA-Z]` 是一个字符类，用于识别大写或小写的单个字母。该字符类外面的括号表示要将匹配的字母作为一组结果。`\1` 表示第一组重复匹配结果的回溯引用（backreference）序号，`+` 号表示匹配的字母可以出现一次或多次。

以上就是匹配模式，接下来用什么来替换呢？我们的目标是将重复的字母减少到最多 2 个，因此解决对象是与 `[a-zA-Z]` 匹配的词汇的一部分。正则表达式中 `[a-zA-Z]` 部分表示一个匹配结果，我们将其定义为 `$1`。这时，我们的替换值就是 `$1$1`。举例来说，在 `yummm` 的匹配中，`[a-zA-Z]` 匹配到第一个 `m`，整个正则表达式匹配到 `mmm`，然后将 `mmm` 用 `mm` 替换掉。要让这个表达式在 `text_microblog` 字段类型中发挥作用，需要添加代码清单 6.3 中的 XML 内容。

#### 代码清单 6.3 定义一个 `charFilter`，使用正则表达折叠重复字母

```
<charFilter class="solr.PatternReplaceCharFilterFactory"
  pattern="([a-zA-Z])\1+"
  replacement="$1$1"/>
```

### 6.3.2 保留主题标签、提及符号和连字符词项

`StandardTokenizer` 处理示例微博时出现了一些小问题。具体来说，`StandardTokenizer` 把主题标签的 `#` 号和提及符号的 `@` 号过滤掉了。此外，`StandardTokenizer` 把连字符词项分开了，例如，`i-Pad` 变成了 `i` 和 `Pad`。这导致查询 `iPad` 时找不到示例微博，正如前一节所见到的。稍后会介绍如何处理 `iPad` 查询问题。现在，让我们讨论一下，为什么要保留主题标签的 `#` 号和提及符号的 `@` 号。

在上一节中，`StandardTokenizer` 将 `@ManningBooks` 的 `@` 号过滤掉了。

6 如果你需要复习一下正则表达式，建议在 [www.regexplanet.com](http://www.regexplanet.com) 上学习和测试正则表达式。

这意味着，我们失去了这个词项的一些信息。具体来说，@ManningBooks 在社会化网络环境中拥有特殊含义，它是 Manning 出版社的专用社交账号。这里的提及符号有特定意图，文本分析时移除 @ 号的做法可能会导致意外的搜索结果，特别是文本分析与词干提取结合的情况下容易产生意外。

主题标签也面临同样的问题。#fail 作为一个主题标签，常用于表达一个人对他人、地方或某种事物（如品牌）的不满意。如果想找到所有包含 #fail 的微博，仅匹配包含 fail 的微博可能有问题，例如 I partied too late last night, I hope I don't fail today's mid-term exam. 因此，在文本分析中保留 #fail 与保留以 # 开头的 fail 是两码事。一般来说，我们希望保留主题标签和提及符号，这样就可以灵活区分包含 #fail 主题的微博和仅包含 fail 一词的微博。这里给出的建议是，在文本分析中需要保留特定词项的上下文语境。

希望你赞同在微博中保留 # 号和 @ 号开头的字符的做法。接下来我们要搞清楚如何去做。首先，我们要明确字符的移除规则。StandardTokenizer 使用字分隔符将文本拆分成词元，其中 # 号和 @ 号也被视为字分隔符之一。如果你是 Java 开发人员，你的初步想法可能会考虑扩展 StandardTokenizer，重写这两个特殊字符的移除行为。不过，对 StandardTokenizer 进行扩展并不容易。更重要的是，无须编写自定义代码就可以解决问题！借此机会，我们学习两个新的 Solr 文本分析工具，即 WhitespaceTokenizerFactory 和 WordDelimiterFilterFactory。

### WhitespaceTokenizerFactory

WhitespaceTokenizerFactory 是一个非常简单的分词器，仅根据空格对文本进行拆分。回忆一下 StandardTokenizer 将示例微博拆分成 23 个独立的分词单元。与之相比，WhitespaceTokenizer 的分词结果有所差异，如图 6.9 所示。

看上去有些许改进？也许吧。这个分词器解决了主题标签、提及符号和连字符问题，但同时又引出了一些新问题。具体来说，表情符号(:)) 现在成了一个分词单元，省略号(...) 与 Beach 连在了一起，即 Beach...。所幸，这些新问题可以使用 Solr 的 WordDelimiterFilterFactory 轻松解决。

### WordDelimiterFilterFactory

WhitespaceTokenizer 简单地根据空格对文本进行拆分，WordDelimiterFilterFactory 则对其进行有效补充，解决了空格拆分导致的大多数问题。该过滤器在更高层面上使用各种解析规则，将分词单元变为子词(subwords)。在详细介绍这些规则之前，让我们先来看看这个过滤器如何帮助我们保留主题标签和提及符号的两个特殊字符。在示例微博上使用代码清单 6.3 中的选项配置 WordDelimiterFilterFactory。

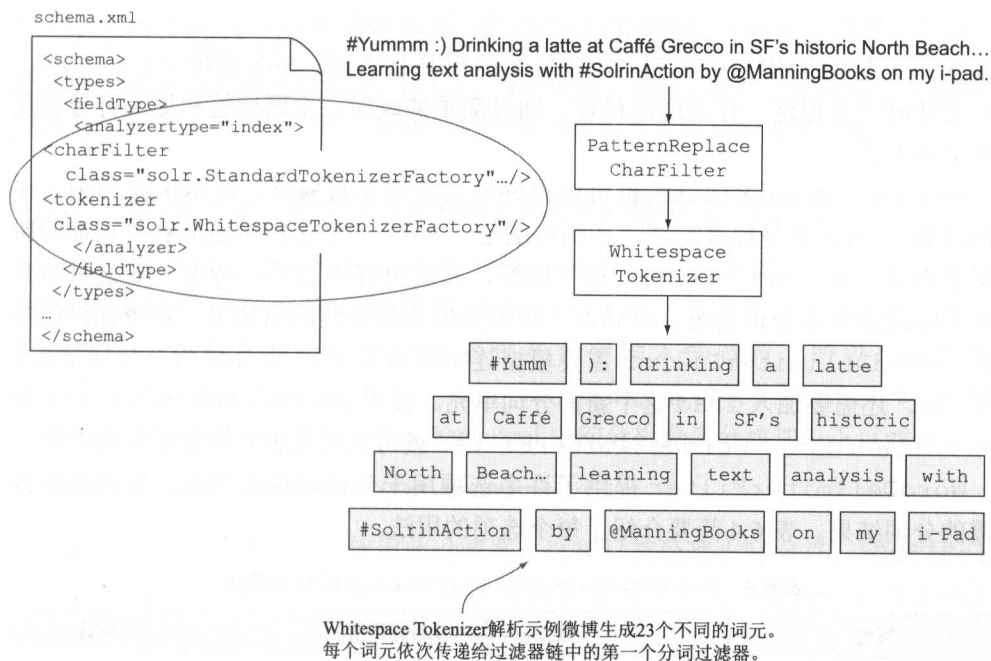


图 6.9 使用 WhitespaceTokenizer 生成的词元仍然是 23 个，但与使用 StandardTokenizer 生成的结果有所不同

#### 代码清单 6.4 定义 WordDelimiterFilterFactory，选项设置为保留 # 和 @

```
<filter class="solr.WordDelimiterFilterFactory"
  generateWordParts="1"
  splitOnCaseChange="0"
  splitOnNumerics="0"
  stemEnglishPossessive="1"
  preserveOriginal="0"
  catenateWords="1"
  generateNumberParts="0"
  catenateNumbers="0"
  catenateAll="0"
  types="wdfotypes.txt"/>
```

自定义一个用于分隔单词的字符类型文件 wdfotypes.txt。

默认情况下，这个过滤器也会过滤掉主题标签的 # 号和提及符号的 @ 号。但与 StandardTokenizer 不同，WordDelimiterFilter 提供了一种简单的方法，使用一个简单的“类型”映射文件（这里是 wdfotypes.txt）来指定哪些字符可以作为文本分隔符。wdfotypes.txt 文件包含两个映射：

```
\# => ALPHA
@ => ALPHA
```



这些设置将 # 号和 @ 号映射到 ALPHA 类,这意味着,WordDelimiterFilter 实例就不会把它们作为分隔符。# 号前面的反斜杠表示, Solr 读取 wdf.ftypes.txt 文件时不会把这一行当成注释行。通过简单的映射,主题标签和提及符号就保留在文本中了。

WordDelimiterFilter 也可以很好地处理连字符问题。在示例微博中,作者使用的 i-Pad 不是标准写法,正确写法是 iPad。如果示例微博能够与查询的所有可能形式(如 iPad、i-Pad)进行匹配,那就再好不过了。考虑一下索引与查询两个阶段的文本分析需求,以确保三种查询形式都能匹配出结果。希望你已经搞清楚,WordDelimiterFilter 需要根据连字符将 i-Pad 拆分成两个分词单元: i 和 Pad, 还需要加入 iPad 这个新的分词单元。设置 generateWordParts=1 和 concatenateWords=1, WordDelimiterFilter 就可以实现想要的确切分词结果。

WordDelimiterFilter 提供了许多选项用于分词转换的微调,以帮你实现所需的分词结果。表 6.4 简要介绍了每个选项的用法。

表 6.4 WordDelimiterFilterFactory 的配置选项

属性	启用(="1")后的作用	默认
generateWordParts	使用内置解析规则与其他选项分隔单词,生成子字词部分	enabled (1)
splitOnCaseChange	驼峰式(Camel-Case)拼写词在解析时根据字母大小写进行分隔。例如, SolrInAction 会被分隔为三个分词单元: Solr、In 与 Action	enabled (1)
splitOnNumerics	字母数字混合词按照数字进行分隔。例如, R2D2 会被分隔为 4 个分词单元: R、2、D 与 2	enabled (1)
stemEnglishPossessive	移除词项中的 's。例如, SF's 变为 SF	enabled (1)
preserveOriginal	除了该过滤器生成的分词之外,保留文本中的原始分词。例如, SF's 应保留为 SF	disabled (0)
concatenateWords	将多个子字词部分连接成一个分词单元。例如, i-Pad 在连字符处分隔成两个分词单元: i 与 Pad,然后将这两个分词单元连接成 iPad,产生第三个分词单元	disabled (0)
generateNumberParts	数值型数据使用标点符号(如双折号)拆分为多个分词单元。例如,电话号码 867-5309 拆分为 867 与 5309	enabled (1)
concatenateNumbers	若数字分词单元被拆分,则将拆分的部分连接起来组成一个新词项;为保持电话号码的参照性,867-5309 会被拆分成三个分词单元: 867、5309 与 8675309	disabled (0)
concatenateAll	当 generateWordParts="1" 且 generateNumberParts="1" 时,将所有部分连接成一个分词单元	disabled (0)

WordDelimiterFactory 在处理内容时需要进行一些试验性尝试。我们建议使用 Solr 的分析表单进行试验,如 6.2.7 节的做法。至此,我们已经解决了主题标签、提及符号和连字符问题,接下来我们把注意力转向重音符号的处理,如 *caffé*。

### 6.3.3 使用 ASCIIIFoldingFilterFactory 移除变音符号

在搜索领域,往往一些简单的做法会让效果大不相同。字符中的变音符处理就是这样一种情况,示例微博的 *caffé* 或 *jalapeño* 包含变音符。大多数情况下,你无法确定用户搜索时是否会输入变音符,所以 Solr 提供了 ASCIIIFoldingFilterFactory 方法,将出现的变音符转换成 ASCII 码等价值,前提是存在对应的 ASCII 码。在 schema.xml 中的分析器部分添加以下过滤器:

```
<filter class="solr.ASCIIFoldingFilterFactory"/>
```

最好在小写过滤器之后使用此过滤器,这样只需处理小写字符。ASCIIIFoldingFilter 仅适用于拉丁字符,其他语种请使用 solr.analysis.ICUFoldingFilterFactory,这个工厂方法在 Solr 3.1 之后的版本可用。

### 6.3.4 使用 KStemFilterFactory 提取词干

词干提取根据特定语言规则,将词转换为基础词形。Solr 提供了许多词干提取过滤器,每种各有优缺点。这里介绍一个基于 Krovetz 词干提取算法的过滤器: solr.KStemFilterFactory。与 PorterStemmer 等其他流行的词干提取器相比,这个词干提取器的转换并不是那么“激进”。这里使用 KStemFilterFactory 对 drinking 和 learning 这样的词项移除 ing。<sup>7</sup>

```
<filter class="solr.KStemFilterFactory"/>
```

表 6.5 展示了使用 KStemFilterFactory 和 PorterStemFilterFactory 进行词干提取的示例。

表 6.5 KStemmer 与 Porter 算法的词干提取比较

原始词项	词干 (KStemmer)	词干 (Porter)
drinking	drink	drink
requirements	requirement	requir
operating	operate	oper
operative	operative	oper

7 参见 R. Krovetz, 1993: “Viewing morphology as an inference process,” in R. Korfhage et al., Proc. 16th ACM SIGIR Conference, Pittsburgh, June 27–July 1, 1993; pp. 191–202.

续表

原始词项	词干 (KStemmer)	词干 (Porter)
wedding	wedding	wed
learning	learning	learn

从这个例子可以看出, 词干提取 Porter 算法比 KStemmer 算法更为激进。过于激进的问题是, 搜索应用可能会匹配出与用户查询无关的文档, 这样会影响查准率。假如用户搜索 wedding in July, 使用 Porter 算法会提取出 wedding 的词干 wed, 而 wed 也是 Wednesday 的缩写。再举一例, 使用 Porter 算法提取 operating 和 operative 的词干都是 oper, 因此包含 covert operative 和 operating system 的文档都将是查询 operating capital 的匹配结果。

这里需要注意, 对示例微博使用 KStemmer 进行词干提取时, drinking 提取的词干是 drink, 但 learning 提取的词干并不是我们所期望的 learn。出现这种奇怪的词干提取结果是因为 KStemmer 使用了一个保护词列表, 列表中的词不会被提取词干, learning 就是其中一个保护词。

一般来说, 词干提取器扩展了与查询匹配的文档集, 即提高了查全率, 但同时会降低查准率。第 14 章会进一步讨论词干提取。

### 6.3.5 在查询阶段使用 SynonymFilterFactory 加入同义词

在词元流中 SynonymFilterFactory 会为重要词项加入同义词。举例来说, 你可能想要在词元流中为 house 添加 home 这个同义词。在大多数情况下, 同义词的添加只用于查询阶段的分析。这样做有助于减少索引的大小, 同义词列表的变更维护也更容易些。如果在索引阶段加入同义词, 你会发现词项对应多出一个新同义词, 针对这个变化, 不得不重新对文档进行索引。如果只在查询处理时加入同义词, 新同义词的添加无须重新进行索引。在 schema.xml 中定义 SynonymFilter 的方法如下:

```
<filter class="solr.SynonymFilterFactory"
  synonyms="synonyms.txt"
  ignoreCase="true" expand="true" />
```

这个过滤器只对查询分析器有效, 特别适用于单个词项的同义词。另外, 你还要考虑它在过滤器链中所处的位置。要确定合适的位置, 需要在同义词匹配之前考虑做怎样的转换。通常最有用的做法是, 将这个过滤器作为查询分析器的最后一个, 以便同义词列表可以认为分词单元上的所有其他转换都已经做完。假设在同义词过滤之后使用 ASCIIIFoldingFilter, 则意味着同义词列表需要包括变音符, 例如, caffè。

在示例微博中，有几个词可以使用同义词，包括 SF、latte 和 caffe。在 Solr 中建立这些词项的同义词，需要在过滤器定义中添加以下的 synonyms.txt 文件：

```
sf,san fran,sanfran,san francisco
latte,coffee
caffe,cafe
```

Solr 提供了映射方法，这虽然看起来不错，但你可能想知道谁会去手动配置上千个同义词？视情况而定吧，这也是 Solr 目前存在的一个问题。目前 Solr 技术社区有一些可用的扩展，例如，将 WordNet 英语同义词数据库转成 Solr 格式，详情参见 <https://issues.apache.org/jira/browse/LUCENE-2347>。

### 6.3.6 把过滤器组合在一起

在示例微博上使用这些过滤器，得到的文本分析结果如图 6.10 所示。

现在让我们回到 Solr 的分析表单，查看之前尝试的查询是否能找到结果。在 6.2.7 节中，我们使用简单的 text\_general 字段类型查询 San Francisco drink cafe ipad，没有找到结果；换成使用 text\_microblog 字段类型，该查询就能找到相关度很高的那条示例微博，如图 6.11 所示。

表 6.6 中查询词项现在能够与文本分析之后的示例微博词项相匹配了。

表 6.6 基于文本分析的查询词项与示例推文匹配

查询词项	文档中匹配的词项	解释
San Francisco	SF's	在查询文本分析中,WordDelimiterFilter 移除了 SF's 的 's, SynonymFilter 添加 SF 作为 San Francisco 的同义词
drink	Drinking	LowercaseFilter 将 D 转换为 d, KStemFilter 移除了 ing
cafe	Caffé	在查询处理中, LowercaseFilter 将 C 转换为 c, ASCIIFoldingFilter 将 é 转换为 e, SynonymFilter 添加 caffe 作为 cafe 的同义词
iPad	i-Pad	WordDelimiterFilter 在连字符处分隔 i-Pad, 通过连接两个分词单元生成一个新的分词单元 iPad

举例说明用途的这个查询看起来有点奇怪。关键在于了解，Solr 提供了丰富的内置文本分析工具，为复杂文本处理提供了灵活的解决方案。最终目的是让用户无须考虑文本的语言差异，能够轻松地使用自然语言，通过搜索应用找到相关文档。

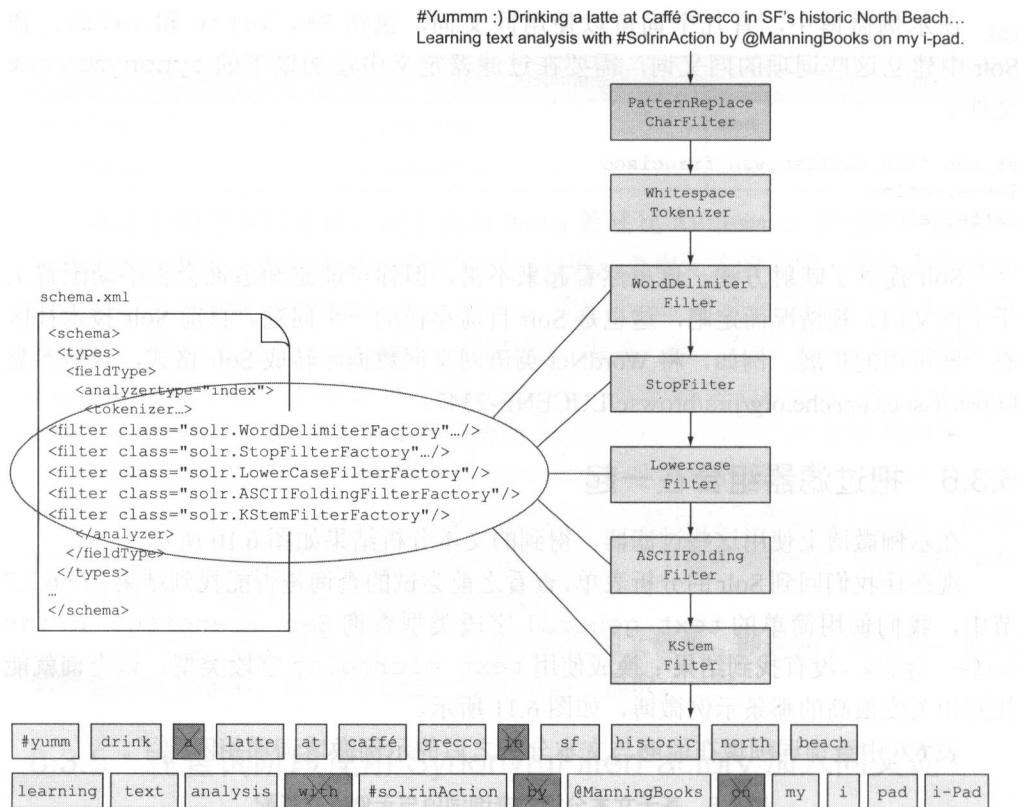


图 6.10 对示例微博使用新的自定义 text\_microblog 字段类型的分析结果

如果你对微博文本分析感兴趣，想要使用 text\_microblog 字段类型进行文本分析，可以使用随书代码中的 ch6/schema.xml，将其配置到本地 Solr 服务器上。更新了 schema.xml 之后，重启 Solr 服务器，然后使用 post.jar 工具将 tweet.xml 文档提交给 Solr 服务器。

```

cd $SOLR_IN_ACTION/example-docs
java -jar post.jar ch6/tweets.xml

```

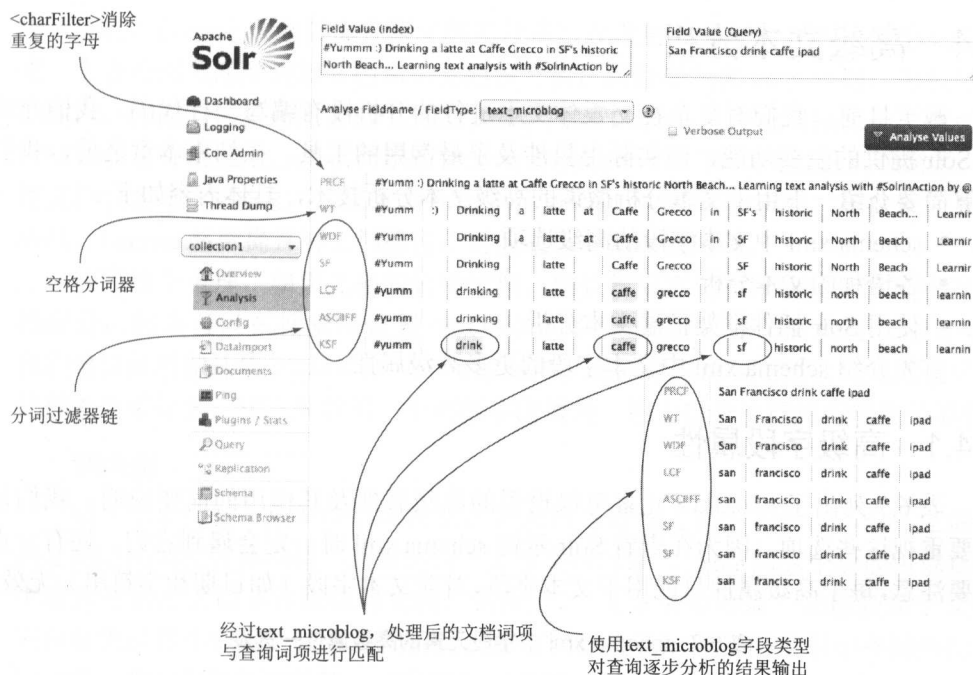


图 6.11 Solr 分析表单显示了对示例微博使用 text\_microblog 字段类型的分析结果

输出结果如下所示：

```
Posting file tweets.xml to http://localhost:8983/solr/collection1/update
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">140</int>
  </lst>
</response>

<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">67</int>
  </lst>
</response>
```

对文档进行搜索之后，你可以搜索 `catch_all:@thelabdude`，以确认这些微博例子是可以被搜索到的。请随意尝试各种查询，看看 Solr 怎么做文本分析。接下来介绍一些高级文本分析主题。

## 6.4 高级文本分析

截至目前，我们对复杂社交媒体文本进行的分析没有编写一行代码。我们介绍了 Solr 提供的主要功能，但实际上只涉及了最常用的工具。在结束本章之前，我们打算简要介绍一下用于文本分析微调的高级文本分析技术，具体内容如下：

- schema.xml 中文本字段的高级选项
  - 各语种的文本分析
  - 使用 Solr 插件框架扩展文本分析
- 首先介绍 schema.xml 中文本字段的更多高级属性。

### 6.4.1 高级字段属性

表 6.7 列出了 <field> 元素可以设置的高级属性及其作用的简要说明。我们认为要重视这些选项，因为在查看 Solr 示例 schema.xml 时一定会遇到它们。还有一点需要注意，每个高级属性只能用于文本字段，对非文本字段（如日期和字符串）无效。

表 6.7 schema.xml 中字段元素的高级属性一览表

属性	启用(="true")后的作用	默认值
omitNorms	禁用该字段的长度规范化与索引阶段权重，这有助于节省索引存储空间。在相关度评分中，规范化赋予较短的文档较小的权重。因此，如果文档长度相当，你应该考虑忽略规范化，这样有助于减少索引大小。如果需要在索引阶段对字段加权，那么不能忽略规范化（omitNorms="false"）。Solr 会将权重值编码到归一值中。对基础类型的字段，如日期、字符串与数值，默认忽略规范化	false
termVectors	Solr 提供类似结果功能，用于找到与特定文档相似的文档。类似结果功能需要启用词向量，这样 Solr 能计算两个文档之间的相似性度量。存储词向量对大型索引而言代价昂贵，因此在确实需要时再启用	false
termPositions	常用于提高 Solr 高亮功能的执行效率，第 9 章将会深入讨论。启用词项位置会增加索引大小	false
termOffset	常用于提高 Solr 高亮功能的执行效率，第 9 章将会深入讨论。启用词项偏移会增加索引大小	false

#### 忽略字段规范化

让我们仔细看一下可选的 omitNorms 属性，这是 Solr 初学者常见的一个疑惑。正如我们在第 3 章中讨论的，规范是基于文档长度规范、文档权重及字段权重三者的浮点值（Java 浮点数）。底层的 Lucene 将这个浮点值编码为单字节，细想一下，这样做很酷。文档长度规范化用来提升短文档。在不涉及太多细节的前提下，相对



于长文档, Lucene 会略微提升短文档的权重, 以改进相关度评分。从理论上讲, 如果一个查询词项匹配到短文档和长文档各一篇, 两篇都包含 `once` 一词, 那么与长文档相比, 匹配的词项在短文档中权重更高, 所以 Lucene 认为短文档比长文档更相关一些。在这种情况下, 词项权重的计算是词项频次 ( $1$ ) 除以文档的词项总数 ( $N$ )。短文档包含 10 个词项, 其权重为 0.10; 长文档包含 1000 个词项, 其权重为 0.001。因此, Lucene 略微提升短文档排位, 对应地以规范形式编码。

按照这个道理, 如果文档的长度相似, 而且没有在索引阶段使用字段提升和文档提升, 那么可以在搜索时设置 `omitNorms="true"`, 以节省内存。刚开始时, 我们建议使用默认值 `omitNorms="false"`。这样做的话, 文档长度规范化可以优化搜索结果排名。接下来看另一个高级字段属性, 用于提升文档相似度的计算性能。

### 词向量

第 3 章介绍过, Solr 使用词向量计算文档与查询之间的相似度。Solr 还提供文档之间的相似度计算功能, 通常称为“更多类似结果”。Solr 的这个功能可以在索引中查找与指定文档非常相似的文档。实质上, “更多类似结果”功能使用了文档的词向量来计算相似度。任意文档的词向量可以在查询阶段使用索引中存储的信息进行计算。为达到更好的性能, 词向量可以在索引时预先计算和存储下来。

利用 `termVectors` 可选属性可以设置每个词向量在索引时存储。因此, 如果你打算在搜索应用中大量使用“更多类似结果”功能的话, 则应为文本字段设置 `termVectors="true"`。在某些情况下, 词向量还可以提升搜索结果高亮功能的速度, 有关搜索结果高亮详见第 9 章。如果在文档索引之后决定启用词向量, 则需重新索引所有文档。

书中其他内容涉及这些属性时再做具体介绍。举例来说, 在第 9 章讨论搜索结果高亮时, 我们会介绍 `termPositions` 和 `termOffsets` 属性。现在, 让我们把注意力转到另一个高级文本分析主题: 多语种分析和语种检测。

## 6.4.2 各语种文本分析

本章讨论的主要是英文文本分析。实际上, 你选择的文本分析方案是服务于特定语言的。英文微博内容的文本分析方案并不适用于德语或法语内容。每种语言都有各自的分词规则、停用词列表和词干提取规则。一般来说, 你需要为索引分析中的每种语言设计一个专门的 `<fieldType>` 字段。因此, 你在本章中学到的许多技术就能派上用场了, 它们仍然适用于英文之外的语言分析。

如何在索引时选择正确的文本分析器呢? 假设你要索引所有的文档, 不考虑索引中的不同语种情况, 一种简单的做法是, 为每个语种分别使用一个唯一字段。假设要在微博搜索应用中索引法语微博, 我们定义以下字段:

```
<field name="text_fr" type="text_microblog_fr"
  indexed="true" stored="true" />
```

从这个字段定义可以看出, 该字段类型是 `text_microblog_fr`, 可以分析法语微博。懂法语的读者可以把 `text_microblog_fr` 字段类型定义作为一个练习。Solr 提供了一个现成的法语分析字段类型 `text_fr`, 如代码清单 6.5 所示。它可以进行基础的法语文本分析, 我们直接以它的使用为例。

代码清单 6.5 Solr 提供的字段类型定义, 用于示例博文中的法语文本

```
<fieldType name="text_fr" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.ElisionFilterFactory" ignoreCase="true"
      articles="lang/contractions_fr.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
      words="lang/stopwords_fr.txt" format="snowball"/>
    <filter class="solr.FrenchLightStemFilterFactory"/>
  </analyzer>
</fieldType>
```

StandardTokenizer 适用于绝大多数拉丁语系。

法语停用词文件。

法语词干提取。

有关 `text_fr` 字段类型需要注意以下几点:

- 由于 StandardTokenizer 适用于拉丁语系的大多数语言, 所以这里使用它进行分析, 但是如果你想保留主题标签和提及符号, 那么需要使用 WhitespaceTokenizer。
- 从 `lang/stopwords_fr.txt` 文件中加载自定义的停用词。
- 对法语微博使用法语专用词干提取器 FrenchLightStemFilterFactory, 因为每种语言的词干提取规则各不相同。

法语微博的文本分析已经有了单独的字段和字段类型, 接下来需要在索引时使用它。如果已经知道文档语种是法语, 在索引文档时就可以手动配置 `text_fr` 字段。举例来说, 下面这条微博援引了伏尔泰的一句经典名言:

*Le vrai philosophe n'attend rien des hommes, et il leur fait tout le bien dont il est capable*

——伏尔泰

代码清单 6.6 是 Solr 索引了这条法语微博的 XML 文档内容。

代码清单 6.6 法语微博示例

```
<add>
  <doc>
    <field name="id">3</field>
    <field name="screen_name">@thelabdude</field>
    <field name="type">post</field>
    <field name="lang">fr</field>
    <field name="timestamp">2012-05-23T09:35:22Z</field>
    <field name="favourites_count">10</field>
    <field name="text_fr">Le vrai philosophe n'attend rien des hommes, et il
      leur fait tout le bien dont il est capable. Voltaire</field>
  </doc>
</add>
```

在索引阶段指明文本语种。

加入 text\_fr 字段，让文本能够使用正确的字段类型分析法语。

这里将该字段的语种设置为 fr 并明确指定 text\_fr 字段类型。如果已经知道文本是法语，这种方法没问题。但是如果事先不清楚文本的语种，那该如何处理？此外，如果内容包含多种语言，要对其进行多语种搜索又该如何处理？第 14 章将介绍多语种搜索，深入探讨 Solr 的多语种分析功能，包括语种检测、特定语种分词、词干提取、字符处理以及多语种内容混合的处理策略。

### 6.4.3 使用 Solr 插件扩展文本分析

如果 Solr 的内置工具无法解决你的文本分析问题，那该怎么办？本章以此问题作为收尾。正如你在本章中所看到的，我们仅使用了 Solr 的内置工具，没有编写任何代码就实现了强大的示例微博内容转换处理。因此，很少会遇到 Solr 内置工具无法解决的文本分析需求，即便遇到，Solr 的插件框架可用于构建应用专属的文本分析组件。

首先，我们需要找到一个 Solr 内置工具无法解决的需求。第 5 章讨论过多值字段，links 字段中索引了零个或多个 URL。微博文本中的所有链接都简化为在线服务商 bitly.com 提供的短网址。例如，Solr 主页网址 (<http://lucene.apache.org/solr/>) 对应的 bit.ly 短网址为 <http://bit.ly/3ynriE>。从搜索的角度看，短网址没有什么用，无法想象有人会在搜索框里输入 bit.ly/3ynriE。当搜索 URL 时，找到的文档包含这个 URL 对应的短网址。也就是说，搜索 lucene.apache.org/solr 会找到包含 <http://bit.ly/3ynriE> 的微博。因此，在索引时我们需要提取短网址，将其替换为对应的完整网址。为获取完整的 URL，我们使用 HTTP 的 HEAD 请求，通过重定向找到完整的 URL。bit.ly 短网址可以使用它的 Web Service API。

好了，清楚了要解决的问题是什么，对问题的解决思路也有了基本认识。接下来需要确定在 Solr 文本分析过程的哪个阶段插入我们的解决办法。我们需要确定构建的插件类型。在 6.2 节中我们了解到，Solr 文本分析包括以下组件：分析器 (Analyzer)、字符过滤器 (CharFilter)、分词器 (Tokenizer) 与分词过滤

器 (TokenFilter)。分析器将分词器和分词过滤器链组合成一个组件。我们不希望因为解决一个 URL 问题而替换掉整个分析器, 而是希望利用已有的分词器和过滤器链。分词器将文本解析为词元流。分词过滤器对分词单元进行转换、替换或移除操作。除非有必要, 否则不要重新定义分词器。

我们的解决方案只涉及使用完整 URL 替换短网址这个分词单元。因此, 针对这个需求, 自定义一个分词过滤器, 这是 Solr 中最常用和最简单的文本分析定制方法。也就是说, 你可以通过类似的处理方法来构建自己的分析器、分词器或字符过滤器。

要创建自定义的分词过滤器, 你需要编写两个具体的 Java 类: 一个类扩展自 Lucene 的 `org.apache.lucene.analysis.TokenFilter` 类, 用于实现过滤; 另一个工厂类扩展自 Lucene 的 `org.apache.lucene.analysis.util.TokenFilterFactory` 类。这个工厂类是必需的, 有了它 Solr 才能根据 `scheme.xml` 文件提供的配置对这个过滤器进行实例化。

### 自定义 TokenFilter 类

首先自定义一个 `TokenFilter`。这里主要关注文本分析的自定义过程, URL 解析的具体实现细节留给有兴趣的读者。代码清单 6.7 给出了短网址解析的自定义分词过滤器类的代码框架。

代码清单 6.7 自定义 Lucene TokenFilter 类, 解析短网址

```
package sia.ch6;

import java.io.IOException;
import java.util.regex.Pattern;

import org.apache.lucene.analysis.TokenFilter;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.tokenattributes.CharTermAttribute;

public class ResolveUrlTokenFilter extends TokenFilter {
    private final CharTermAttribute termAttribute =
        addAttribute(CharTermAttribute.class);
    private final Pattern patternToMatchShortenedUrls;

    public ResolveUrlTokenFilter(TokenStream in,
        Pattern patternToMatchShortenedUrls) {
        super(in);
        this.patternToMatchShortenedUrls = patternToMatchShortenedUrls;
    }

    @Override
    public boolean incrementToken() throws IOException {
        if (!input.incrementToken())
            return false;
    }
}
```

自定义过滤器扩展自 `TokenFilter`。

自定义 `TokenFilterFactory` 知道如何构造 `TokenFilter`。

调用方法, 在分词流中处理每个词元。

```

char[] term = termAttribute.buffer();
int len = termAttribute.length();

String token = new String(term, 0, len);
if (patternToMatchShortenedUrls.matcher(token).matches()) {
    termAttribute.setEmpty().
        append(resolveShortenedUrl(token));
}
return true;
}

private String resolveShortenedUrl(String toResolve) {
    // TODO: 对短网址进行解析
    return toResolve;
}
}

```

分词结果是一个短网址，对其进行解析和替代。

具体实现方式留给有兴趣的读者。

如果选择此过滤器方案，我们建议你考虑此方案对索引性能的影响。如果大量文档需要索引，比如说社交媒体内容容量就很大，那么实施方案需要解决网址解析带来的高延迟。一种初步的解决方案是使用分布式缓存，如 memcached，以避免多次解析网址，并充分利用短网址服务商如 bitly.com 提供的 Web Service API。通常，API 方式允许在单个请求中对多个短网址进行批处理，这比使用 HTTP HEAD 请求重定向来解析网址的办法更有效。

### 自定义 TokenFilter 类

为 Solr 开发自定义 TokenFilter，还需要编写一个工厂类，实现 TokenFilter 的实例化。此工厂类负责接收 schema.xml 中指定的属性，并将它们转换成 TokenFilter 创建所需的参数。schema.xml 中分词过滤器的定义方法如下：

```

<filter class="sia.ch6.ResolveUrlTokenFilterFactory"
    shortenedUrlPattern="http://bit.ly/[w-]+" />

```

此工厂类使用 shortenedUrlPattern 属性，为文本分析中的短网址匹配问题编译一个 Java 的 Pattern 对象。此处的例子只支持 bit.ly 网址，在实际应用中通过扩展正则表达式，以支持更多的短网址来源。

接下来，需要考虑在 text\_microblog 字段类型的过滤链中哪个位置上放置此过滤器。在解决短网址问题之前无须对短网址进行任何转换。所以我们认为它应该紧跟 WhitespaceTokenizer 之后，在 WordDelimiterFilter 之前。这个工厂类的 Java 实现如代码清单 6.8 所示。

代码清单 6.8 通过自定义 Lucence TokenFilterFactory 创建 TokenFilter

```

package sia.ch6;

import java.util.Map;
import java.util.regex.Pattern;

import org.apache.lucene.analysis.TokenFilter;
import org.apache.lucene.analysis.TokenStream;
import org.apache.lucene.analysis.util.TokenFilterFactory;

public class ResolveUrlTokenFilterFactory extends TokenFilterFactory {

    protected Pattern patternToMatchShortenedUrls;

    @Override
    public ResolveUrlTokenFilterFactory(Map<String,String> args) {
        super(args);
        assureMatchVersion();
        String shortenedUrlPattern = require(args, "shortenedUrlPattern");
        patternToMatchShortenedUrls =
            Pattern.compile(shortenedUrlPattern);
    }

    @Override
    public TokenFilter create(TokenStream input) {
        return new ResolveUrlTokenFilter(input,
            patternToMatchShortenedUrls);
    }
}

```

自定义类必须扩展 TokenFilterFactory，以便 Solr 知晓如何实例化该工厂类。

重写 init 方法，访问 schema.xml 中工厂类提供的属性。

重写 create 方法，返回 TokenFilter 的完整配置实例。

仅需几行代码就自定义了一个 TokenFilter！这段代码的核心是，Solr 进行文本分析时，将此过滤器作为 schema.xml 的过滤器定义与自定义 TokenFilter 的配置示例两者之间的中介。

最后要将包含插件类的所有 JAR 文件放在 Solr 初始化时能够找到的位置。为简单起见，我们建议再新建一个 plugins/ 目录，将此目录位置添加到 solrconfig.xml 中，具体做法参见第 4 章。

```
<lib dir="plugins/" regex=".*\.jar" />
```

启动 Solr 服务器，plugins 目录中的所有 JAR 文件都可以通过 Solr 的 ClassLoader 加载。如果 Solr 在初始化期间无法找到你的自定义类，请尝试使用 plugins 目录的完整路径。

## 6.5 本章小结

祝贺你！你已经掌握了一些难度较大的内容。至此，你应该已经系统化掌握了 Solr 的文本分析和索引过程。回顾一下，本章一开始介绍了通过字段类型定义来进

行基础的文本分析。这让我们了解到，非结构化文本字段的字段类型可以在索引创建和查询处理两个阶段使用一个分析器，或分别使用两个独立的但相互兼容的分析器。每个分析器由一个分词器和分词过滤器链组成。为测试文本分析效果，我们通过 Solr 的分析表单，对示例微博使用 StandardTokenizer 和过滤器链，实现了停用词移除和小写转换。

测试结果表明，基础的文本分析不足以处理示例微博内容分析的所有问题。因此，我们使用了 Solr 的其他一些内置工具来满足更复杂的需求。具体来说，我们使用 PatternReplaceCharFilterFactory 编写了一个简单的正则表达式，将词项中的重复字母（如 yummm、soooo）减少到两个以内。我们使用 WhitespaceTokenizer 和 WordDelimiterFilter 保留了微博中的主题标签 # 号和提及符号 @ 号。WordDelimiter 也被证明能够很好地处理连字符词项，例如，搜索 iPad 会匹配出包含 i-Pad 的文档。我们还学习了词干提取和同义词扩展方法，用以提升搜索引擎的匹配能力。总体来说，我们仅使用 Solr 的内置工具，没有编写一行代码，就实现了一个强大的微博文本分析方案。

本章最后介绍了一些高级文本分析主题。具体来说，我们介绍了字段的高级属性，例如，如果不需要在索引阶段提升权重或对字段进行规范化，设置 omitNorms="true" 来减少索引对内存和存储空间的要求。随后，我们了解了 Solr 应对多语种的内置语种检测方法，更多详细内容请参见第 14 章。最后，我们编写了一个自定义 TokenFilter 来解析短网址。本章想要传达的一个核心理念是，针对各类需求，在 Solr 中可以轻松地自定义分析器、分词器、分词过滤器及字符过滤器等。

在下一章中，我们将学习如何使用 Solr 查询和处理搜索结果。



## 第2部分

# Solr的核心功能

接下来的 6 章将介绍 Solr 的核心功能，帮助你实现面向用户的强大搜索和发现体验。

搜索最重要的特征是为查询提供相关的搜索结果。在第 7 章中，你将学习如何构造复杂的查询，如何对搜索结果排序和浏览结果，以及如何以各种格式返回搜索结果。

关键词搜索是 Solr 的主要功能，除此之外，大多数搜索应用还需要其他功能来改善整体的用户体验。在第 8 章至第 11 章中，我们将讨论 Solr 最常见的一些附加功能。

具体来说，我们将讨论：分面，用于搜索结果优化；搜索结果高亮，提供文本片段，显示匹配到的关键词的上下文信息；拼写检查和自动建议，帮助那些打字太快或拼写出错的用户修正查询；字段折叠与结果分组，去除多个类似结果，以体现结果文档的多样性。

如果第 8 章至第 11 章的某些内容不适用于你的当前需要，在阅读时可以跳过它们。例如，搜索结果分组是许多搜索引擎的一个常见功能，假如你的数据不需要这个功能，就可以跳过第 11 章。

最后，第 12 章将深入介绍在生产环境中启用与扩容 Solr 时需要考虑的重要因素，提供了一份多年 Solr 实践的第一手经验总结。可扩展的搜索是 Solr 的 DNA 中不可或缺的一部分，这一章对 Solr 核心功能进行了充分总结。

# 7

## 执行查询和处理搜索结果

### 本章要点

- 介绍 Solr 搜索功能的各种请求处理器
- 通过 Solr 可插拔式搜索组件扩展搜索结果
- 通过查询解析器组合实现强大的查询功能
- 返回的查询结果包括静态值和动态值
- 根据值、函数与相关度进行排序
- 调试搜索结果

第 5 章和第 6 章主要介绍了 Solr 的索引与文本分析功能。正如你所见，文本分析发生在索引创建和执行查询两个阶段。接下来两章主要关注的是，根据文本内容分析生成倒排索引，以及后续的搜索使用。本章切换到查询端，探索 Solr 的搜索功能。

本章会回顾请求处理器的概念（参见第 4 章），以及 SearchHandler 的环境配置，它是 Solr 最重要的请求处理器。SearchHandler 运行一个或多个 SearchComponent，包括 QueryComponent，它用于响应搜索请求，执行主查询。在讨论 QueryComponent 时，我们还会介绍 Solr 的许多查询解析器，展示 Solr 的强大查询语法与搜索功能。

当熟悉了完整的查询语法之后，我们会介绍如何处理返回的搜索结果。本章会介绍如何对搜索结果进行排序、分页、返回特定字段和动态生成值、定义搜索结果

格式及调试搜索请求等。本章涵盖了 Solr 的核心搜索功能，介绍了构建 Solr 驱动的复杂搜索应用需要的所有基础知识。为了深入学习 Solr 的核心搜索功能，让我们先回顾一下 Solr 对传入的请求如何处理。

## 7.1 Solr请求详解

Solr 最常见的请求类型是在 Solr 索引中查找相关文档的查询 (query)。除此之外，Solr 还可以处理许多不同类型的请求。第 4 章曾介绍过，所有的请求（例如，文档更新和查询）基本上都是通过请求处理器提交给 Solr。搜索处理器 (search handler) 是查询处理的默认请求处理器，通过调用一个或多个搜索组件，每个组件处理搜索请求的一部分，从而满足查询各个阶段的要求。例如，通过搜索组件执行主查询，其中分面（详见第 8 章）、搜索结果高亮（详见第 9 章）和拼写检查（详见第 10 章）都有各自的搜索组件。要让查询请求能够使用主搜索组件，需要通过一个或多个查询解析器对查询文本进行解析，其作用是理解查询语法，将其映射为适当的查询对象集，以便在 Solr 索引中找到相关文档集。本节在第 4 章介绍的请求处理器和搜索组件基础上，进一步剖析 Solr 的请求，讨论请求处理器、搜索组件与查询解析器之间的交互。我们首先从请求处理器入手。

### 7.1.1 请求处理器

请求处理器 (Request Handler) 基本上是 Solr 所有请求的入口。它的作用是接受请求，执行某些功能，向客户端返回结果。Solr 拥有许多请求处理器来完成各项任务，例如，搜索执行 (SearchHandler)、从一台服务器向另一台服务器复制索引 (ReplicationHandler)，以及添加新文档以更新 Solr 索引等 (UpdateRequestHandler)。你还可以通过 LukeRequestHandler 获取更多有关 Solr 索引的丰富信息，通过 SystemInfoRequestHandler 获取内存使用情况和 Solr 设置等服务器信息等。为简单起见，大多数请求处理器继承自 RequestHandlerBase 这个 Java 类，但这种做法并不是强制性的。SolrRequestHandler 接口实现的任何类都可以作为一个请求处理器。虽然我们可以编写自己的请求处理器作为 Solr 的插件，实现 SolrRequestHandler 接口，但是大多数 Solr 用户还是会使用 Solr 内置的请求处理器。图 7.1 显示了大多数 Solr 内置请求处理器的继承关系。

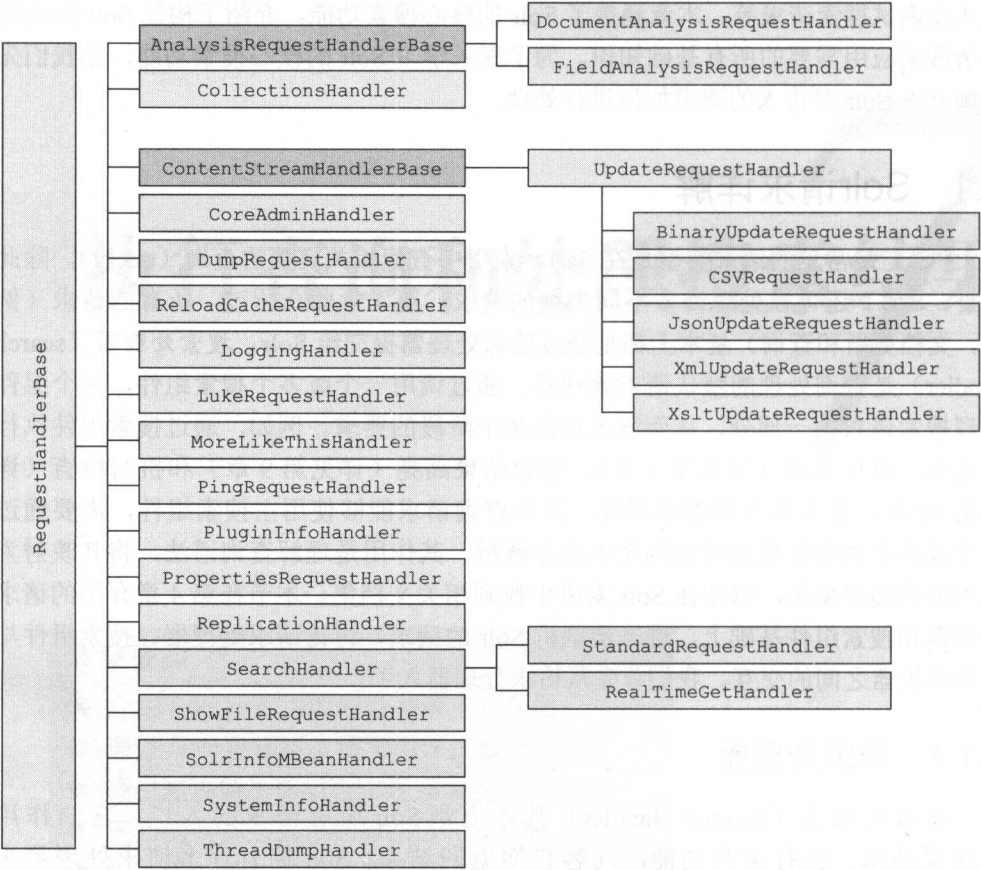


图 7.1 各个 Solr 内置请求处理器的类层次关系, 在 Solr 默认部署时一并提供。AnalysisRequestHandlerBase 和 ContentStreamHandlerBase 是两个抽象类, 所以不能直接引用。少数不是继承自 RequestHandlerBase 的内置请求器没有在这里列出

从图 7.1 中可以看出, Solr 可以处理许多不同类型的请求。SearchHandler 作为 Solr 最常用的请求处理器, 它是搜索请求处理的默认请求处理器。图中有一些 Solr 内置处理器会在后续章节中详细介绍, 其他处理器则留给你自行探索。表 7.1 对每个请求处理器的常见用法做了简要介绍。

表 7.1 Solr 多种请求处理器的简要说明。除非另有说明, 所有处理器都位于 org.apache.solr.handler 包

请求处理器类名	说明
DocumentAnalysisRequestHandler	接受文档流如 UpdateRequestHandler, 但返回的是处理过的内容, 而不是将其添加到 Solr 的索引。用于整个文档内容的文分析调试。文档分析参见第 6 章

续表

请求处理器类名	说明
FieldAnalysisRequestHandler	与 DocumentAnalysisRequestHandler 类似,但是它处理的是一个或多个字段,而不是整个文档。用于特定字段的内容分析调试。文本分析参见第 6 章
UpdateRequestHandler	接受文档流,对其进行处理,并添加到 Solr 索引。一些请求处理器继承自 UpdateRequestHandler,可以处理各种内容类型,如 XML、JSON、CSV、二进制输入以及 XSLT 转换的内容。如果直接使用它(而不是使用它的子类),UpdateRequestHandler 会尝试根据内容类型为每个请求选择正确的子类。向 Solr 发送更新参见第 5 章
CollectionsHandler	用于管理 SolrCloud 集合(参见第 13 章)
CoreAdminHandler	用于获取详细信息,并在运行的 Solr 实例中管理多个 Solr 索引或内核(参见第 12 章)
DumpRequestHandler	用于验证/调试发送到 Solr 的内容。如果是批量数据,例如向 Solr 发送一组文档,DumpRequestHandler 可显示包含该数据的内容流,用于调试目的
ReloadCacheRequestHandler	该请求处理器位于 FlatFileSource 类里。当 ExternalFileField 关联的文件发生改变时,该处理器会被用来重新加载缓存。ExternalFileField 允许在 Solr 索引以外的外部文件中存储与更新字段信息,但需要在外部文件更新时通知 Solr。当外部文件更新时,让 ReloadCacheRequestHandler 通知 Solr(参见第 15 章)
LoggingHandler	在输出日志信息时,记录并控制使用哪些日志记录器及其日志记录级别
LukeRequestHandler	报告 Solr 索引的元信息,包括词项数量、使用了哪些字段、索引中的热门词项及索引中的词项分布。另外,还可以以文档为单位,请求相关信息
MoreLikeThisHandler	基于输入的文档找到文本相似的文档。在用户表达了兴趣的文档上可以进行内容推荐(参见第 16 章)
PingRequestHandler	基于 Solr 内核执行查询情况与 Solr 服务器上存在的运行状况检查文件,返回 OK 状态消息。这常用于负载均衡器探测 Solr 服务器停止服务的端点(Endpoint)。如果运行状况检查文件存在,且 Solr 可以成功执行查询,则表明服务器正常;如果运行状态检查文件被删除,服务器报告已宕机,但仍允许非用户进行查询,则会影响故障转移能力。Solr 管理界面包含一个启用/禁用链接,这让服务器的服务切换变得简单(参见第 12 章)
PluginInfoHandler	提供在运行的 Solr 实例中已经加载和使用的插件信息
PropertiesRequestHandler	返回所有的系统属性值。如果指定参数名,则返回与该参数名匹配的相关属性

续表

请求处理器类名	说明
ReplicationHandler	一个 Solr 内核（从）使用该处理器从另一个 Solr 内核（主）获取索引信息。它允许多台 Solr 服务器维护同一个 Solr 索引的多个副本，并保持它们同步，以实现多台服务器之间达到负载均衡（参见第 12 章）
ShowFileRequestHandler	用于返回存储在 Solr 文件系统中的文件。常用于返回 schema.xml、solrconfig.xml 及其他配置文件
SolrInfoMBeanHandler	用于返回 Solr 实例中 SolrMBean 所有可用对象的状态信息，包括大多数核心 Solr 对象信息，通过该处理器无须使用 JMX 就可以创建 Solr 监视
SearchHandler	Solr 处理搜索的默认请求处理器。该处理器包括插件式搜索组件（参见 7.1.2 节），可以在单个搜索请求中启用多种搜索功能。SearchHandler 包括 StandardRequestHandler（已弃用）和 RealTimeGetHandler。由于 RealTimeGetHandler 在 Solr 接收新内容和索引可用之间需要花费一些时间，所以在内容提交到 Solr 索引之前，该处理器根据更新日志可以在搜索请求中实时检索新内容。本章会详细介绍 SearchHandler
SystemInfoHandler	提供了 Solr 实例操作系统时间点概览，包括 Solr/Lucene 版本信息、JVM 信息（如内存使用率与 JVM 版本）、JMX 以及操作系统信息等
ThreadDumpHandler	提供 Java 线程转储，列出当前 JVM 中正在运行的线程

虽然表 7.1 中的许多请求处理器不会被大多数 Solr 用户使用到，但了解一下这些请求处理器，可能会在未来有所帮助。每个请求处理器使用之前都要在 solrconfig.xml（详见第 4 章）中进行定义和配置。代码清单 7.1 是一个配置示例，启用了更新文档的 UpdateRequestHandler、从一台服务器向另一台服务器复制文件的 ReplicationHandler，以及几个用于搜索的 Searchhandler 实例。

代码清单 7.1 在 solrconfig.xml 中启用请求处理器

```
<config>
...
<requestHandler name="/select" class="solr.SearchHandler" />
<requestHandler name="/update" class="solr.UpdateRequestHandler">
<requestHandler name="/replication"
class="solr.ReplicationHandler"
startup="lazy" />

<!-- The below entries are for demonstrative purposes -->
<requestHandler name="/private/search " class="solr.SearchHandler" />
...
</config>
```

定义一个请求处理器时，需要指定两个属性：`name` 和 `class`。Solr 维护了一个请求处理器查找列表，根据每个请求的指定要求，派发对应的请求处理器。`class` 属性对应特定的 Java 类（`SolrRequestHandler` 接口实现），使用指定名称的请求处理器处理查询请求。

如果请求处理器的名称以反斜杠 / 开头（这是标准做法），`name` 就使用该请求处理器的相对 URL。举例来说，使用代码清单 7.1 的配置启动 Solr 示例程序（如果不知道怎么操作，请回顾第 2 章），你会得到以下 URL：

- `http://localhost:8983/solr/collection1/select/`
- `http://localhost:8983/solr/collection1/update/`
- `http://localhost:8983/solr/collection1/replication/`
- `http://localhost:8983/solr/collection1/private/search/`

这样做的话，其中一些请求处理器会报错，这是因为它们需要根据请求指定默认参数，不过你仍然可以通过添加它们的 `name` 到 Solr URL 来调用这些请求处理器。

对于代码清单 7.1，还需要补充两点。首先，请注意，请求处理器 `/select` 和 `/private/search` 定义的类相同。这样做没有问题，它会使用相同的类来创建两个不同的请求处理器。其次，请注意，在 `/replication` 的请求处理器上有一项 `startup="lazy"` 属性。一般来说，某个请求处理器不会用在所有的 Solr 服务器上。设置启动选项为 `lazy`，这表示，直到第一次调用该请求处理器时，Solr 才会加载它。这样做会节省资源，但会导致该请求处理器的初次查询变慢。

回顾一下之前的章节，你会发现已经使用到了多个请求处理器。在第 4 章中，你已经见过 `UpdateHandler` 和 `SearchHandler` 的使用，例如，向 Solr 添加文档并对这些文档进行搜索。后续章节会继续讨论其他请求处理器，在这里你应该已经很好地了解了请求处理器是什么以及如何定义它们。了解请求处理器有助于掌握搜索组件，由于偶尔存在功能上的重合，所以有时容易将搜索组件与请求处理器相混淆。

### 7.1.2 搜索组件

上一节介绍了许多不同类型的 Solr 请求处理器，`SearchHandler` 是执行搜索的默认响应处理器，搜索默认会返回什么呢？是否只包括搜索结果？有没有匹配到的最相关类目，或是对匹配到的文档的部分文本进行高亮呢？如果只匹配到很少结果或没有结果，可供选择的拼写建议是否可以作为默认请求的一部分返回呢？

从上一节可知，许多搜索功能可以通过发送单独的、可用的请求来调用多个请求处理器来实现。理想情况下，向 Solr 发送一个请求，就能得到所有预期的信息。这正是搜索组件存在的原因。



搜索组件是在搜索处理器生命周期内发生的可配置的处理步骤。搜索组件让搜索处理器将实现单个搜索请求的可重用的功能组合链接在一起。

搜索组件在 `solrconfig.xml` 中进行配置，具体做法参见第 4 章。为帮助你回顾之前内容，代码清单 7.2 演示了如何创建搜索处理器及一系列默认搜索组件。

代码清单 7.2 搜索处理器的搜索组件默认列表

```
<searchComponent name="query" class="solr.QueryComponent" />
<searchComponent name="facet" class="solr.FacetComponent" />
<searchComponent name="mlt" class="solr.MoreLikeThisComponent" />
<searchComponent name="highlight" class="solr.HighlightComponent" />
<searchComponent name="stats" class="solr.StatsComponent" />
<searchComponent name="debug" class="solr.DebugComponent" />

<requestHandler name="/select" class="solr.SearchHandler">
  <arr name="components">
    <str>query</str>
    <str>facet</str>
    <str>mlt</str>
    <str>highlight</str>
    <str>stats</str>
    <str>debug</str>
  </arr>
</requestHandler>
```

每个搜索处理器可以将定义的任何搜索串联起来。

代码清单 7.2 的内容都是 Solr 的默认配置，也就是说，它可以出现在任何地方，甚至可以脱离 `solrconfig.xml` 的上下文来看。对于理解如何配置搜索组件，启用新的搜索组件，以及理解搜索处理器的默认工作原理，这些都是有益的。

在第一个 `<searchComponent />` 标签中定义了两个属性：一个是 `name`，另一个是 `class`。`name` 的值是 `query`，`class` 的值是 `solr.QueryComponent`。该搜索组件只需定义一次，之后它可以被任意数量的请求处理器调用。在 `/select` 请求处理器中，你会看到一个 `arr` 元素，其属性 `name` 为 `components`。该元素包含了搜索处理器定义的搜索组件列表。`components` 部分的每一行对应一个搜索组件名称，既可以是默认的搜索组件，也可以是 `solrconfig.xml` 中定义的其他搜索组件。

### 请求处理器和搜索组件的默认设置

请求处理器和搜索组件通常从以下两种来源接收请求设置：在 Solr 的 URL 中传递的代码，以及查询字符串变量。在 `solrconfig.xml` 中配置一个请求处理器或搜索组件时，通过设置变量自动添加到每个请求，就好像它们在 Solr 的 URL 中传递一样。

代码清单 7.3 为搜索组件添加了两个部分: `invariants` 部分和 `defaults` 部分。如果 Solr 的请求 URL 里没有重写值, `defaults` 部分就是默认值。如果想要执行安全默认设置, 查询搜索组件的 `q=*` 确保不会发生意外或避免没有结果返回的情况, 这种情况下 `defaults` 默认值就是有用的。

`invariants` 部分与 `defaults` 部分类似, 只不过 Solr 请求 URL 中常量参数无法被重写。强制执行某些安全过滤器或执行代码清单 7.3, 以确保所有请求的结果返回数是 25 (`rows=25`), 关键词搜索的默认字段是 `df=content_field`。这种情况下, `invariants` 设置就是有用的。你会看到 `solrconfig.xml` 中许多请求处理器和搜索组件都会使用类似的 `defaults` 和 `invariants` 设置。

代码清单 7.3 "query" 组件的默认配置

```
<searchComponent name="query" class="solr.QueryComponent">
  <lst name="invariants">
    <str name="rows">25</str>
    <str name="df">content_field</str>
  </lst>
  <lst name="defaults">
    <str name="q">*</str>
    <str name="indent">true</str>
    <str name="echoParams">explicit</str>
  </lst>
</searchComponent>
```

大多数搜索组件支持在它们的 XML 配置中设置相应的属性。如果希望对默认搜索组件执行此操作, 请将搜索组件的名称设置为默认名称中的一种 ("query"、"facet"、"mlt"、"highlight"、"stats" 或 "debug"), 另外, 你还可以更改搜索组件的默认配置。

即使默认搜索组件没有在 `solrconfig.xml` 中明确定义, 它们也是默认存在的。在先前的代码清单就替换了默认配置。还可以通过添加 `first-components` 和 `last-components` 部分来插入搜索组件, 这两部分表示分别在搜索处理器执行的组件列表的开始或结尾处插入附加组件, 请参见第 4 章的 4.2 节来回顾如何在搜索处理器中添加搜索组件。

在搜索处理器的所有搜索组件中, 查询组件是最重要的。因为它负责查询的初始化执行和搜索结果的特定格式响应, 而且随后的其他搜索组件需要在它的基础上执行。查询组件使用查询解析器对搜索处理器请求中的传入查询进行解析。Solr 支持多种查询解析器, 下一节会具体介绍。

### 7.1.3 查询解析器

查询解析器将 Lucene 查询解释成搜索语法, 以便查找所需的文档集。Solr 支持多种查询解析器, 还支持用户编写自己的查询解析器。正如搜索组件专门用于单个请求处理器 (SearchHandler), 查询解析器也是专门用于单个搜索组件 (QueryComponent) 的。图 7.2 展示了这种关系: SearchHandler 执行一个 QueryComponent, QueryComponent 使用 QueryParsers。

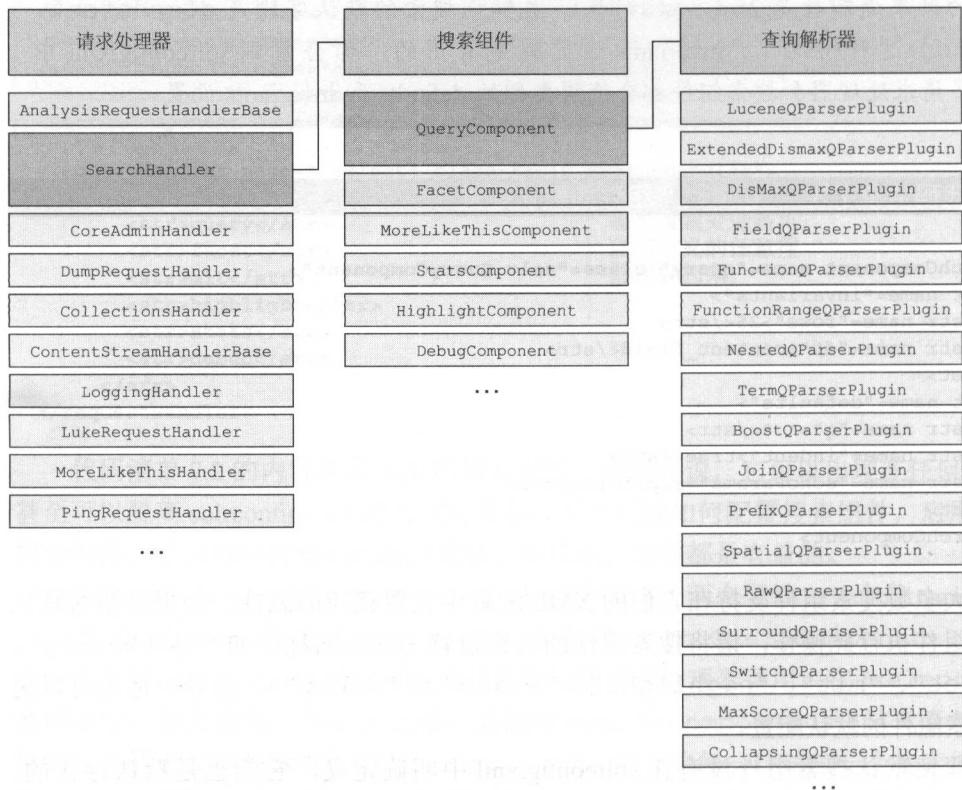


图 7.2 QueryComponent 使用的查询解析器, 搜索组件专门负责执行 SearchHandler 中指定的查询。查询解析器的类实现为 QParserPlugin 类

从图 7.2 可以看出, Solr 提供了许多查询解析器, 每个查询解析器实现为一个 QParserPlugin 类。如果已有的查询解析器无法实现你需要的查询解析功能, 可以编写自己的 QParserPlugin。Solr 最常用的查询解析器是 Lucene 查询解析器 (LuceneQParserPlugin) 和 eDisMax 查询解析器 (ExtendedDismaxQParserPlugin), 详情参见 7.4 节和 7.5 节。图 7.2 的其他查询解析器将在 7.6 节介绍。在深入了解查询解析器的工作原理之前, 先来了解一下各种查询解析器的使用方法, 这有助于后续的学习。

## 7.2 查询解析器的使用

执行搜索时，QueryComponent 根据查询解析器传递的值处理初始的用户查询（q 参数）。上一节提到，LuceneQParserPlugin 是 Solr 的默认查询解析器。该默认查询解析器可以很容易进行替换，也可以在同一个查询里与多个查询解析器组合使用。每个查询解析器执行各自的查询类型，支持对应的查询语法，对应不同的使用场景，具体参见 7.4 节至 7.6 节。本节介绍查询解析器的使用方法，包括如何更改默认查询解析器、如何组合查询解析器，以及如何配置查询解析器。

### 7.2.1 指定查询解析器

QueryComponent 的默认查询解析器类型可以使用搜索请求中的 defType 参数进行修改：

```
/select?defType=edismax&q=...  
/select?defType=term&q=...
```

修改默认查询解析器类型会对处理 q 参数的查询解析器进行更改，从而导致查询结果的变化。除了修改默认查询解析器类型之外，还可以使用 Solr 的特殊语法修改查询中使用的查询解析器：

```
/select?q={!edismax}hello world  
/select?q={!term}hello  
/select?q={!edismax}hello world OR {!lucene}title:"my title"
```

在第三个例子中，一个查询调用了两个不同的查询解析器（edismax 和 lucene）。与使用 defType 参数相比，这是在查询中指定查询解析器的优势所在。Solr 包含了许多这样有用的查询解析器，本章会对它们做具体介绍。

{!...} 语法通过行内定义查询解析器来调用 Solr 的某一项功能，这称为局部参数，下一节会详细介绍。

### 7.2.2 局部参数

局部参数为特定上下文提供定制化请求参数。通常的做法是在 URL 里向 Solr 提交请求参数，但有时也可以在查询内的特定部分指定一些参数。在一个查询中，局部参数可以只为特定查询解析器传递请求参数，而不是全局传递所有请求参数。上一节已经介绍过在查询内部修改查询解析器。局部参数不仅可以修改查询解析器，还可以修改任何请求参数。

## 局部参数语法

局部参数是一组用来表示请求参数的键值对集合，只限于当前上下文有效。其语法如下：

```
{!param1=value1 param2=value2 ... paramN=valueN}
```

局部参数以 `{!` 开头，以 `}` 结尾，包含以空格分隔的键值列表，其中键值由等号分隔。例如，

```
/select?q=hello world&defType=edismax&qf=title^10 text&q.op=AND
```

等价于下面使用局部参数的查询：

```
/select?q={!defType=edismax qf="title^10 text" q.op=AND}hello world
```

这两个查询的本质区别在于：第一个例子中所有请求参数都是全局的，因此它们可以用于请求中的任何位置。在局部参数的例子中，`defType`、`qf` 和 `q.op` 参数只在 `q` 参数范围内使用。如果需要在查询中多次使用不同配置的一个查询解析器，使用局部参数就可以做到。

我们可以在局部参数中指定 `defType` 参数，也可以使用 `type` 参数改写 `defType` 参数。从字面上看，`defType` 表示默认类型，也就是说，它为所有查询指定了默认查询解析器类型。在局部参数中使用 `type` 参数可以修改特定上下文里的默认类型：

```
/select?q={!type=edismax qf="title^10 text" q.op=AND}hello world
```

`type` 参数仅在局部参数中使用，如果要在顶层定义一种类型作为默认类型，则必须使用 `defType` 参数。与之相对，`defType` 参数既可以用于请求层次，又可以用于局部参数。

由于一些局部参数值可能包含特殊字符（空格、引号等），所以需要单引号或双引号将局部参数括起来，或者需要对特殊字符进行转义。上一条查询的 `qf` 参数中包含空格，所以使用双引号将它括起来，有关 Solr 的特殊字符转义的更多内容，详见 7.4.1 节末尾。

你可能已经注意，7.2.1 节与本节中使用局部参数语法指定查询解析器的做法有细微差异。在本节中，我们指定 `{!type=edismax ...}`；之前我们使用了更简单的 `{!edismax...}`。如果只指定一个值，则 `type` 会被视为默认局部参数的键，所以这两种方式都是有效的。如果关键词搜索不指定字段的话，它会在默认字段上搜索，也有可能局部参数中传递值，将它用于 `type` 默认局部参数的键。因为局部参数的语法比较短，所以查询解析器的定义一般使用 `{!queryParserName}` 语法，其

他局部参数则必须使用完整的 `{!key1=value1 key2= value2 ...}` 语法。

### 参数值

局部参数声明的值就是要传递给查询解析器的值。下面的局部参数声明中，向查询解析器传递的值是 `hello world`：

```
/select?q={!edismax qf="title^10 text"}hello world
```

如果不在局部参数声明之后指定值的话，而在局部参数中定义参数值有时会更容易些。局部参数的特殊键 `v` 就是专门用于处理此种情况。因此，之前的查询可以另外定义为：

```
/select?q={!edismax qf="title^10 text" v="hello world"}
```

值得注意的一点是，在局部参数中将 `query` 值放入 `v` 参数中时，只需留意特殊字符的转义。举例来说，如果要搜索带双引号的 `"hello world"`，则以下两个查询是等价的：

```
/select?q={!edismax qf="title^10 text"}"hello world"  
/select?q={!edismax qf="title^10 text" v="\\"hello world\\""}"
```

在局部参数中明确定义查询值会带来一些复杂性，为了提升请求语法里的参数重用性，常常会用到参数解引用（`parameter derefencing`）。

### 参数解引用

参数解引用提供了查询中任意变量的替换方法。这个功能类似于 SQL 的参数查询，可以不使用查询语法来单独定义查询输入。引用参数的语法如下：

```
/select?q={!edismax v=$userQuery}&userQuery="hello world"
```

在这个例子中，`userQuery` 参数作为用户定义的查询字段串变量传递给 Solr，而 Solr 本身不能理解该参数。通过将 `eDisMax` 查询解析器的传入值指定为解引用参数 `$userQuery`，该值可以放在请求的其他位置。初次使用可能会觉得其发挥作用有限，但由于在 `solrconfig.xml` 中为每个请求处理器和查询组件使用默认值、追加值或常量定义了默认参数（参见第 4 章），你可以为自己的搜索应用指定一组请求参数集替换预配置。



### 减少参数解引用的重复

如果需要以不同方式重用部分查询，并且不希望在请求中复制它们，那么参数解引用可以派上用场。将解引用参数链接在一起是可行的，一个解引用参数可以由其他解引用参数组成。第 15 章的 15.1.4 节介绍了一个例子，本地营业税作为一个参数传入，用于计算商品文档的总成本，然后对其进行排序，以字段值返回等多种操作。

现在，你应该已经掌握了如何更改查询解析器，如何通过全局参数和局部参数向查询解析器传递值。接下来，在深入了解 Solr 查询解析器工作原理之前，接下来了解一下查询和过滤器的工作原理。

## 7.3 查询和过滤器

在介绍 Solr 查询解析器工作原理之前，了解用户查询和过滤器的工作原理有助于进一步学习。查询与过滤器的区别是什么，它们之间如何交互，以及最终如何影响搜索请求的性能和质量。

Solr 的搜索主要由两个操作组成：找到与请求参数相匹配的文档；对这些文档进行排序，返回最相关的匹配文档。默认情况下，文档根据相关度进行排序。这意味着，找到匹配的文档集之后，需要另一个操作来计算每个匹配文档的相关度得分。第 3 章介绍过，在 Solr 的倒排索引中匹配文档的查找过程和文档相关度得分的默认算法。

### 7.3.1 fq 和 q 参数

为有效地查找匹配的文档和计算文档的相关度得分，Solr 会用到两个参数：fq 和 q。fq 参数表示过滤器查询，q 参数表示查询。初看这两个参数可能不太好区分，因为相同的查询语法传递到这两个参数中，会返回相同数量的文档。因此，许多搜索请求中使用单个 q 参数。但是理解这两个参数之间的差异，可以更高效地进行搜索。

#### 相关度影响

那么，q 参数与 fq 参数之间的区别是什么？fq 只有一个功能：对匹配的文档进行查询限定。

而 q 参数有两个功能：

- 对匹配的文档进行查询限定。
- 提供相关度算法以及用于相关度评分的词汇列表。



因此, `q` 参数可视为一个特殊的过滤器, 告诉 Solr 在相关度计算时应考虑哪些词项。鉴于这种差异, Solr 使用者倾向于将用户输入的关键词放入 `q` 参数中 (例如, `keywords:"apache solr"`), 将机器生成的过滤器放入 `fq` 参数中 (例如, `category:"technology"`)。

### 缓存和执行速度

从主查询中分离出过滤器查询有两种用途。首先, 过滤器查询通常在不包含任意关键词的搜索之间可以重复使用。因此, 可以考虑将过滤器查询的结果缓存在过滤器缓存中, 参见第 4 章 4.4.2 节的相关讨论。其次, 由于相关度评分操作必须对文档匹配的查询 `q` 中每个词项进行计算, 那么将查询的一部分拆分成过滤器查询 `fq`, `fq` 参数这部分就无须进行额外的相关度计算。这样处理之后, 查询中可作为过滤器的查询部分为相关度评分节省了许多工作。

### 指定多个查询和过滤器

最后, 查询和过滤器还需注意一点: Solr 请求中可以添加任意多个 `fq` 参数, 但只能包含一个 `q` 参数。例如, 两个 Solr 查询 `q=keywords:solr&fq=category:technology&fq=year:2013` 与 `q=keywords:solr&fq=category:technology AND year:2013` 会以同样的次序返回相同的文档。除了 `fq` 参数的缓存用途 (每个 `fq` 参数可以独立缓存), 使用多个 `fq` 参数在功能上等价于将这些参数组合成一个 `fq` 参数。以下小节中介绍的许多查询解析器都能使用 `q` 和 `fq` 参数, 我们要考虑相关度和缓存影响, 根据实际情况选择使用哪一种参数。

### 执行顺序

由于查询和过滤器都会查找文档集并对它们进行操作, 所以出现一个常见问题: 查询和过滤器的执行顺序是怎样的? 一些资料说首先执行过滤器, 另外一些资料说首先执行查询, 还有一些资料说查询和过滤器同时执行。到底情况如何? 这个问题很复杂, 需要视具体情况而定。

从技术层面上看, 操作顺序如下:

1. 在过滤器缓存中对每个 `fq` 参数进行查找。若存在, 缓存的 `DocSet` 将被返回, 以 `OpenBitSet` 进行封装, 其中索引里的每个文档对应一个二进制位 (0 或 1), 以表示该文档是否包含在过滤器中。
2. 若没有在过滤器缓存中找到 `fq` 参数, 但缓存已启用, 那么该过滤器将对索引进行过滤, 得到一个新的 `DocSet`, 这样就对其进行缓存。
3. 所有过滤器的 `DocSet` 做交集 (AND 操作), 得到一个 `DocSet`。
4. `q` 参数与过滤器的 `DocSet` 一起传入, 作为一个 Lucene 查询进行搜索。执行查询时, Lucene 对查询与组合过滤器进行搭桥处理, 将查询与过滤器对

象统一成一个当前的内部 ID（一个整数）。若查询结果和过滤器结果对象包含相同的 ID，则收集该 ID，处理过程包括为匹配的文档计算相关度得分。

5. 如果文档包含任何后置过滤器（下一节将讨论），它们将作为收集过程的一部分，在查询与过滤器做了交集处理之后执行，仅作用于同时匹配组合查询和组合过滤器的文档。

根据这个解释，当缓存启用时，过滤器会先于主查询执行。查询和过滤器随后在收集过程（搭桥步骤）中同时执行，后置过滤器作为一种特殊的过滤器，在查询和过滤器已经找到同时匹配的文档之后使用。图 7.3 是示例搜索请求的查询和过滤器处理的 5 个步骤。

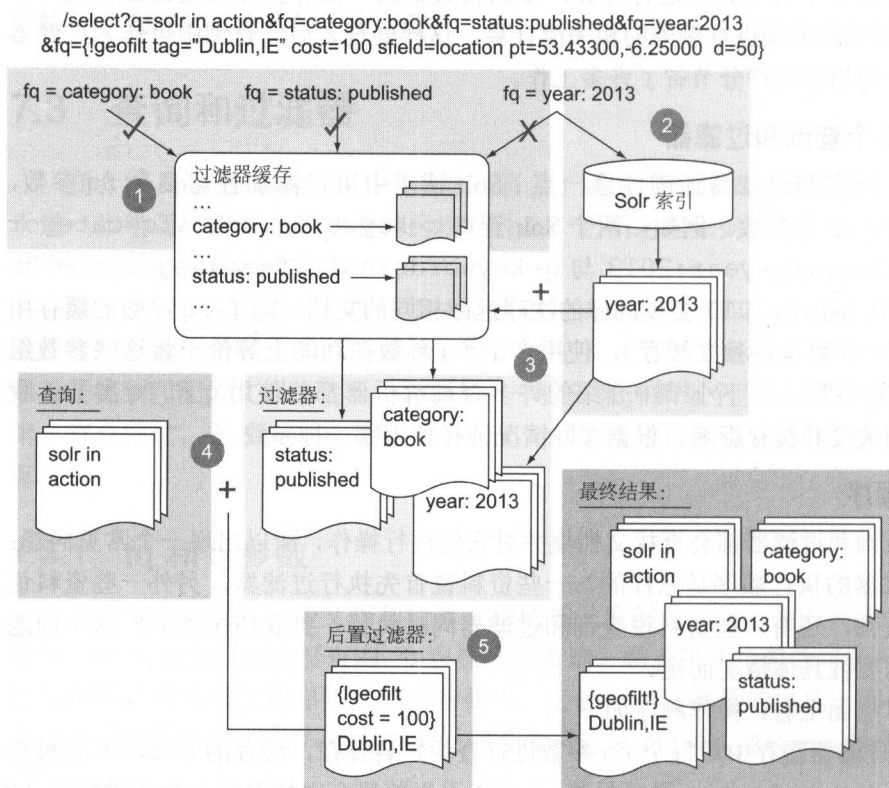


图 7.3 查询和过滤器的组合过程

图 7.3 的每个编号步骤解释如下。① 在过滤器缓存中查找过滤器；② 为缺失的过滤器对索引进行；③ 将每个过滤器组合起来；④ 使用搭桥将查询与已经组合的过滤器进行再组合；⑤ 使用执行成本最高的后置过滤器。最后一步返回最终的 DocSet。如果需要对 Solr 搜索结果进行排序、检索及返回最相关的文档，则根据该查询（而非过滤器）计算相关度。

这看起来很复杂，确实如此。Solr 很好地将这种复杂性隐藏了起来。不过，理解这个过程有助于对使用代价过高的过滤器进行性能调优。Solr 提供更为精细的控制，能够指定哪些过滤器需要进行缓存，以及过滤器的执行顺序，包括在主查询之前、之后或同时进行。下一节将介绍如何通过缓存的开关来控制过滤器的执行情况，指定过滤器的执行顺序，决定是否在查询或其他已执行的过滤器之后执行。

### 7.3.2 处理代价过高的过滤器

对过滤器进行缓存和绕过查询中过滤器部分的相关度处理，这样可以大大节省处理时间。然而，并非所有的过滤器情况都一样。如果尝试对搜索结果进行指定经纬度的地理半径过滤（参见第 15 章的 15.2 节），由于都要涉及数学计算，因此这个过滤器的计算成本可能很高。此外，如果要为数百万个位置生成不同的过滤器的话，如此之多的半径过滤器可能很难进行缓存。在某些应用中可能需要生成许多唯一过滤器，例如，对唯一 ID 进行过滤，造成过滤器缓存过载，导致常用的过滤器会被删除或搜索预热时间会过长。对于这种情况，Solr 能控制哪些过滤器应该缓存，以及确定过滤器的执行顺序。

#### 关闭过滤器缓存

在某些情况下，许多过滤器不需要进行缓存。由于过滤器数量有固定上限，如果最常用的过滤器始终处于缓存状态，则 Solr 实例的性能最佳。为防止不重要的过滤器造成缓存过载，可以使用以下语法关闭哪些过滤器。

```
fq={!cache=false}id:123&  
fq={!frange l=90 u=100 cache=false}  
scale(query({!v="content:(solr OR lucene)"}),0,100)
```

上面的过滤器中，第一个是专门为每个文档生成的过滤器，因此该过滤器不合适放在过滤器缓存中。第二个过滤器也是专用的，尝试找到与查询 `content:(solr OR lucene)` 相关度最高的前 10% 的文档。通过获取该查询的相关度得分，将所有文档的得分按比例置于 1 到 100 区间，然后过滤出 90 到 100 区间之外的文档。由于该过滤器包含一个变量输入（`query` 函数的输入），所以它适合关闭过滤。根据需要可以添加任意多个过滤器，`fq` 参数支持过滤器缓存的开启和关闭。为每个 `fq` 参数指定 `cache=true` 或关闭局部参数缓存，`cache` 默认为 `true`。

#### 改变过滤器执行顺序

如果搜索请求包含多个过滤器，它们的执行顺序会对查询速度产生显著影响。从一般逻辑上讲，能让结果集减少最多的过滤器应最先执行，因为面对文档越少，

过滤器执行速度越快。同样道理, 执行复杂计算的过滤器(例如, 地理空间过滤器在一定半径范围内进行过滤)应考虑靠后执行。它们处理的文档越少, 所耗费的计算资源也就相对少一些。对于需要花费更多代价的过滤器, 通过定义该过滤器相关的执行成本, Solr 允许它们靠后执行。提供过滤器成本的语法如下:

```
fq={!cost=1}category:technology&
fq={!cost=2}date:[NOW/DAY-1YEAR TO *]&
fq={!geofilt pt=37.773,-122.419 sfield=location d=50 cost=3}&
fq={!frange l=90 u=100 cache=false cost=100}
  scale(query({!v="content:(solr OR lucene)"}),0,100)
```

执行成本越高的过滤器, 它的执行应该越靠后。举例来说, category:technology 过滤器的执行成本最低, 所以最先执行。该过滤器执行速度快, 能在单个文档类别中显著减少文档数量, 所以说它的执行成本最低。第二个过滤器(执行成本为2)将所有结果限定在去年的某一天。第三个过滤器是 geofilt 操作, 它计算出一个半径范围, 将找到的结果限定该半径的 50 英里以内。这是最耗费资源的一个操作。第四个过滤器执行成本是 100, 由于数学计算导致耗费资源高, 加之关键词输入具有很大的不确定性, 因此指定其 cost=100 和 cache=false。执行成本从 3 一下跳到 100, 这看起来有些奇怪。执行成本不一定是连续的, 彼此之间只是相对顺序。大于或等于 100 的执行成本会启用 Solr 的一个特殊功能——后置过滤。

### 后置过滤

在一些情况下, 过滤器的执行成本会非常高, 你会希望所有其他查询和过滤器执行之后再执行它。Solr 提供了一种特殊类型的过滤器, 称为后置过滤器。此过滤器在查询和过滤相交处理之后使用。

回顾一下 7.3.1 节, 查询与组合过滤器一起执行(搭桥处理), 找到的每个文档与查询和过滤器都要匹配。后置过滤器是一种特殊过滤器, 仅用于被调用的文档, 让执行成本较低的过滤器先执行, 对整体结果进行限定, 执行成本较高的后置过滤器最后执行, 需要处理的文档数量就少了很多。为过滤器定义 cost 参数, 这也是将一个过滤器转换成后置过滤器的方法。执行成本大于或等于 100 的过滤器都被视为后置过滤器, 使用后置过滤器接口执行。

Solr 的后置过滤器不一定适用于所有类型的查询和过滤, 它只适用于那些使用 PostFilter 接口的查询和过滤。FRange 查询(参见 7.6.3 节)是具有后置过滤器功能的一种查询类型。另外, 也可以编写插件来执行后置过滤器接口, 在主查询和过滤器执行之后使用自定义的过滤器。

理解了查询器与过滤器的明确差异, 以及相关度和性能考虑, 接下来介绍 Solr

中最常用的查询解析器的工作原理。

## 7.4 默认查询分析器 (Lucene 查询解析器)

本书目前介绍的大多数查询都使用了标准的 Solr 语法。这种语法是 Solr 最常见的，由默认查询解析器负责处理。Solr 的默认查询解析器是 Lucene 查询解析器 (LuceneQParserPlugin 类实现)，虽然它是 Solr 的特定类，但还是会在名称上让人感到疑惑。Lucene 查询解析器全面支持 Lucene 语法及 Solr 的一些专用扩展。

### 7.4.1 Lucene 查询解析器语法

第 3 章以间接方式介绍了 Lucene 查询解析器的大部分语法，本节将全面介绍 Lucene 查询解析器支持的查询操作。需要注意，语法必须严格遵守执行。如果查询与语法不完全匹配，会抛出查询异常，导致请求失败。在第 3 章中你已经学到了一种语法，支持许多操作，诸如字段和非字段搜索、必备词项、可选词项、短语搜索、组合表达式、词项邻近度、排除词项、区间搜索、通配符搜索及布尔表达式。本节详细介绍每一种功能如何使用 Lucene 查询解析器的正确方法。

#### 字段搜索

在 Solr 索引中搜索一个值时，一般来说是在特定字段上进行查找。字段搜索的语法是：字段名称加该字段的搜索表达式，中间用冒号分隔，举例如下。

```
title:solr
title:"apache solr" content:(search engine)
```

尽管关键词搜索不明确指定字段的做法很常见，但需要注意，一般在定义的默认字段上进行关键词搜索。举例来说，如果 content 定义为默认字段(df=content)，则以下两个查询是等价的：

```
solr
content:solr
```

还需要注意的是，字段和冒号后面的表达式范围必须明确定义。以下两个查询是等价的（假设 df=content），不过在第一个查询中用户可能有其他意图。

```
title:apache solr
title:apache content:solr
```

如果要在同一字段中搜索多个词项，使用组合表达式，在字段搜索中指定词项的范围：

```
title:(apache solr)
```

如果尝试短语搜索，使用引号（而不是括号）来定义短语范围，尽管这样会导致查询要求短语的所有词项必须同时出现。

```
title:"apache solr"
```

### 必备词项

为指定一个或多个词项必须出现，使用一元运算符 `+` 来连接词项。除非文档包含指定的词项，否则不予匹配。如果匹配的文档必须包含多个词项，使用二元运算符 `AND` 或 `&&`，或者对每个词项都使用一元运算符 `+`。

```
+solr
apache AND solr
apache && solr
+apache +solr
apache solr (假设默认运算符是 AND)
```

如果默认运算符是 `AND` (`q.op=AND`)，在没有指定其他运算符的情况下，每个词项都要求必备。由于每增加一个必备词项会进一步限制文档集中的结果总数，因此通过使用多个必备词项可以加快查询速度，从而进一步优化结果数量。

### 可选词项

相比限制必备字段的做法，扩大匹配的文档数量则适用于另外一些情况。默认运算符是 `OR` (`q.op=OR`)，除非有其他指定，否则每个表达式都是可选的。同样地，多个表达式之间使用二元运算符 `OR` 或 `||`，这表示匹配的文档中至少包含其中一个词项。其语法如下：

```
apache OR solr
apache || solr
apache solr (假设默认运算符是 OR)
```

值得注意的是，可选词项越多会导致文档集越大，`OR` 运算比其他布尔运算的执行成本更高。对于关键词搜索，如果内容数量有限，而且希望以牺牲查准率为代价，确保能够返回一些结果（更高的查全率），那么一般会考虑使用 `OR` 作为默认运算符。由于多个可选词项的文档匹配通常会导致较高的相关度得分，使用 `OR` 作为默认运算符并根据相关度得分排序的话，仍然有可能获得搜索结果中最相关的那部分结果。不过，与要求匹配所有关键词不同的是，扩展查询会得到更多一些奇怪的匹配结果。



## 短语搜索

如果想要匹配彼此相邻的多个词项, 使用引号把它们括起来视为一个短语, 例如:

```
"apache solr"  
"apache software foundation"
```

此查询表达式不能保证匹配出完全一样的文本, 被搜索字段可能包含对短语中词项进行修改的分析器。例如, 搜索 "Raining Cats and Dogs", 如果被搜索字段采用激进式词干提取, 将 and 和 with 视为停用词进行移除, 那么就有可能匹配到 rain cat with dog。话虽如此, 使用引号把词项引起来确实可以保证在连续的词项位置上找到它们, 最合理的特定短语搜索不应该匹配出无关的短语。短语搜索适用于内容中特定字词和多词名称的处理。

## 组合表达式

为处理任意复杂的布尔子句, Solr 使用括号将查询表达式组合在一起, 例如:

```
(apache AND (solr OR lucene)) AND title:(apache solr)
```

组合表达式可以设置表达式的上下文, 例如, 指明在同一字段中搜索多个单词。组合表达式可以任意嵌套。

## 词项邻近度

之前介绍过在引号中包含多个词项来定义短语搜索。实际上, 这是词项相似度搜索的简化版本。通过添加波浪线和词项位置距离数, 搜索位置相近的词项, 不一定是彼此相邻的, 如下所示:

```
"apache solr"~3  
"open source software"~5
```

你可以将短语搜索看成隐含距离为 0 的邻近搜索。以下两个查询是等价的:

```
"apache software foundation"  
"apache software foundation"~0
```

第二个查询的意图是查找与短语 "apache software foundation" 精确匹配的所有文档。该查询中词项之间的位移为 0。同样的道理, 如果指定位移数值为 2, 则查询会匹配 "apache foundation software"、"software apache foundation"、"apache [otherWord] [otherWord2] software foundation", 以及其他短语形式。这些短语可视为对原始短语 "apache software foundation" 进行不超过 2 个位置的词项移动。参照第 3 章所讨论的, 两个词项交换位置相当于移动了 2 个位置。



指定足够大的有效邻近值，可以匹配出文档中任意位置的词项，这与 AND 查询效果类似。假设文档包含的词项少于 1 000 000 个，以下两个查询返回的文档数量相同。

```
apache AND solr
"apache solr"~1000000
```

词项邻近度查询还有一个有趣的副作用是，在文档中词项越靠近，该邻近查询对应的相关度得分越高。第一个查询不关心词项的位置，第二个查询虽然关心词项位置，但会超出文档中的词项数量范围，所以这两个查询的唯一差别是相关度得分及计算成本。词项之间的距离计算与布尔查找相比，花费成本更高。这两个查询返回的文档数量相同。使用邻近度权重可以提升词项更靠近的文档的相似度，这种方法将在第 16 章进一步讨论。

### 字符邻近

不仅可以在词项之间进行邻近搜索，还可以对词项中的字符进行基于编辑距离的搜索，找到拼写相似的词项。字符邻近搜索的语法与词项邻近搜索类似，由于字符邻近处理的是一个词项，所以不带引号：

```
solr~1
supercalifragilisticexpialidocious~5
```

第一个查询找出词项 solr 的编辑距离为 1 的相关词项，如 sol、sor、slr、salr、olr 等。第二个查询为“supercalifragilisticexpialidocious”这样一个更长更复杂的词找出编辑距离最大为 5 的相关词项。在一些搜索应用中，有结果返回好于没有结果，那么对相同词项使用编辑距离，这样就可以看看拼写相似的词项有没有结果返回。这样做可以帮助用户纠正拼写错误，当然还有更好的方法来纠正拼写错误，随后第 10 章会具体介绍。

### 排除词项

有时我们需要从查询中明确排除特定词项。在表达式上使用一元运算符-(减号)或在表达式之间使用 NOT 布尔运算符来排除词项：

```
solr -panel
solr NOT panel
solr AND NOT (panel OR electricity)
-badterm
```

前三个例子的目的是将拼写错误 solar (solar 与太阳有关，一般指太阳能) 导致的无关文档分离出来。正确的拼写应该是 solr，一种搜索引擎。这些例子演示了-和 NOT 运算符的使用方法，第三个查询展示了如何在多个词项中使用 NOT 运算符。

最后一个例子在 Solr 中执行纯粹的排除查询，其等价于 `*: * -badterm`。该查询找到的文档不包含指定排除的表达式。

### 区间搜索

有时候我们不希望查询表达式只匹配出一个值，而是希望匹配出值的整个区间。区间可以是数值区间（例如，价格区间在 20 美元到 25 美元之间）、日期区间（例如，去年修改的所有文档）或字符串区间（例如，app 和 apple 之间的相似拼写，如 appetite）。区间搜索能够找到指定的一组值，其语法为字段名加冒号再加一个方括号，如下所示。

```
number:[12.5 TO 100]
date:[2013-11-04T10:05:00Z TO NOW-1DAY]
string:[ape TO apple]
```

值得注意的是，日期格式必须使用祖鲁时间 (Zulu time) 格式或使用日期运算来指定，否则会抛出异常。如果没有指定区间的最大值和最小值，则需要对开区间的上限或下限使用通配符 (\*)，如下所示。

```
number:[* TO 0]
number:[100 TO *]
date:[NOW-1Year TO *]
```

更甚者，区间的上限和下限都可以打开：

```
field:[* TO *]
```

如果区间搜索不限定区间的上限和下限，看起来会很愚蠢，但这样可以找到字段所有取值的相关文档，这就好像字段的值存在于某两个值之间。在 Solr 的新近版本中也可以通过 `field:*` 来定义。在这种情况下，区间搜索可能是矫枉过正了。

使用方括号可以实现闭区间搜索，使用花括号可以实现开区间搜索：

```
number:{0 TO 100}
```

在这个例子中，如果字段类型是整数型，匹配到的最小值是 1，最大值是 99。方括号和花括号可以混合使用。以下两个查询在整数字段上结果相同：

```
number:[1 TO 100]
number:{0 TO 99}
```

Solr 的区间搜索对许多操作都有用，例如，围绕特定地点绘制边界 (`latitude:[min TO max] AND longitude:[min TO max]`)，或根据日期区间生成各种查询限定，等等。

## 通配符搜索

有些情况下用户需要对 Solr 索引中单词或短语的变体进行匹配。对于用户输入的大多数关键词而言, 词干提取这类技术让通配符搜索变得没那么必要了, 然而对查找以特定字符集开头的文档或替代单个字符的操作, 通配符搜索还是有用武之地。通配符查询的语法包括你要查找的文本, 星号 (\*) 表示一个或多个字符, 问号 (?) 用于替换单个字符。在短语中可以加入任意多个通配符:

```
hel* w?rld, t??s is awe*m?
```

该短语会匹配出 "hello world, this is awesome"。需要注意的是, Solr 能够以非常快的速度进行关键词搜索, 在倒排索引中直接对词项进行精确查找。在通配符搜索中, Solr 必须扫描索引以查找更多与通配符查询相匹配的词项, 这意味着, 在通配符之前包含的字符越多, Solr 能够扫描和在搜索中使用的词项就越少。要了解通配符的性能影响以及如何在搜索词项中较早使用通配符的优化方法, 请参见第 3 章的 3.1.7 节。

## 权重表达式

权重表达式会在第 16 章详细介绍, 这里仅介绍其语法:

```
(apache^10 solr^100 is^0 awesome^1.234) AND (apache lucene^2.5)^10
```

如果表达式后面指定了一个插入号 (^), 无论是词项、短语还是组合表达式, 都可以调整相关度权重。如果你清楚一些表达式比另一个表达式更重要, 或者想为查询的不同方面分配一定量的相关度, 权重表达式这时就能派上用场。

## 特殊字符转义

Solr 中有些字符是保留字符, 也就是说, 它们被当作查询语法进行解析, 而不作为搜索词项。Solr 的这些字符包括:

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : /
```

在一些查询中需要将保留字符作为搜索词项的一部分。举例来说, 尝试执行查询 `q=content:(I'm so happy!!!:))`, Solr 会返回以下结果:

```
org.apache.solr.search.SyntaxError: Cannot parse 'content:( I'm so
happy!!! : ) )': Encountered " ")" " " at line 1, column 30. Was expecting
one of: <BAREOPER> ... "(" ... "&*" ... <QUOTED> ... <TERM> ... <PREFIXTERM>
... <WILDCARD> ... <REGEXPTERM> ... "[" ... "{" ... <LPARAMS> ... <NUMBER>
...
```

如果需要搜索保留字符, 必须将保留字符用引号括起来, 或者使用反斜杠对其进行转义:

```
q=content:"I'm so happy!!! : )"
q=content:(I\'m so happy\\!\\!\\! \: \))
```

在第一个例子中, 查询被引号括了起来, 这表示将其全部视为一个短语, 而不是单独的词项。这样做改变了查询的本意, 可能不太适合多数情况。将每个词项用引号括起来, 也可以达到反斜杠的效果:

```
q=content:("I'm" "so" "happy!!!" ": ")
```

如果要预处理所有词项, 这样做没问题, 但是这种做法带来的工作量很大, 通常是不实际的。这种做法也无法处理双引号的情况, 双引号也是转义字符之一。因此, 关键词中处理保留字符的推荐方法是, 在传入 Solr 之前去除没有搜索价值的保留字符 (也有可能被字段分析器处理过了), 或者对它们依次使用反斜杠进行转义。

正如你所见, Lucene 查询解析器语法支持任意复杂的查询。不过, 对许多用户而言, 查询语法需要严格执行, 任何特殊字符或形式不规范的查询都会导致返回异常, 得不到期望的搜索结果。虽然 Lucene 解析器可以很好地处理以编程方式生成的查询, 但对用户输入的关键词查询也不能做到完美解析, 除非你自行对这些查询进行预处理。所幸, Solr 提供了一个非常棒的查询解析器来处理用户输入的查询, 即 eDisMax 查询解析器。

## 7.5 处理用户查询 (eDisMax查询解析器)

正如上一节谈到, Lucene 查询解析器语法支持创建任意复杂的布尔查询, 但还有一些缺陷, 它不是用户查询处理的理想解决方案。这里面最大的问题在于, Lucene 查询解析器的语法要求严格, 一旦破坏就会抛出异常。指望用户在输入关键词时能够理解 Lucene 查询语法并始终能输入完美的查询表达式, 这显然是不合理的。这也意味着, Lucene 查询解析器在许多搜索应用中对用户不够友好。

Lucene 查询解析器的另一个缺点是, 它不能默认搜索多个字段。df 参数定义了查询解析器默认搜索哪个字段, 但是如果想要以不同权重对多个字段进行搜索呢? 例如, 默认搜索 title 字段和 content 字段, 其中 title 字段的相关度权重高一些。要使用 Lucene 解析器执行此操作, 必须先对用户查询进行预处理。如果需要在单个字段中对所有关键词进行匹配, 则将 `q=some keywords` 转换为 `q=title:(some keywords) OR content:(some keywords)`。如果需要两个字段的关键词都出现文档中, 则将 `q=some keywords` 转换为 `q=(title:some OR content:some) AND (title:keywords OR content:keywords)`。对大多数 Solr 开发人员来说, 这样的查询预解析工作量过大。为了将用户查询直接传入 Solr 并优雅地进行处理, 扩展的析取最大化查询解析器 eDisMax 应运而生。

### 7.5.1 eDisMax 查询解析器概述

eDisMax 查询解析器实际上是由 Lucene 查询解析器和 DisMax 查询解析器组成。DisMax 查询解析器是 eDisMax 查询解析器的旧版本，它只接受关键词和少数几个基本的布尔运算，允许在多个字段中搜索关键词。因为 DisMax 查询解析器是 eDisMax 查询解析器的一个子集，所以不建议使用原始的 DisMax 查询解析器，推荐使用较新的扩展版本。因此，我们不会单独介绍 DisMax 查询解析器，它的大多数功能请参考 eDisMax 查询解析器的介绍。

虽然 eDisMax 查询解析器不是 Solr 的默认查询解析器，但它具有查询语法容错性，不像 Lucene 查询解析器那样严格。对于那些从用户那里直接获取关键词的搜索应用而言，eDisMax 是最佳选择。下一节介绍 eDisMax 查询解析器的常见查询参数。

### 7.5.2 eDisMax 查询参数

eDisMax 查询解析器支持 Lucene 查询解析器的所有查询语法。它们之间只有一个明显差异：eDisMax 对无效的输入语法不会抛出异常，而是会将无效的输入作为文本字符串进行搜索。它还在语法解析上具有一定的容错性，支持特殊关键词，例如，可以理解小写转换后的 AND 和 OR。这种灵活性和容错性让它比 Lucene 查询解析器更适合处理用户输入。

### 7.5.3 搜索多个字段

除了安全地处理用户输入文本和自由地解析查询语法，eDisMax 查询解析器最实用的一个功能是对多个字段进行搜索。eDisMax 查询解析器不是强制将所有可搜索的内容复制到一个默认的 content 字段，而是将每块内容放在各自的字段里，例如 title 字段、description 字段和 author 字段。使用 Lucene 查询解析器的话，就必须为 Solr in Action 构造出如下查询：

```
((title:solr) OR (description:solr) OR (author:solr)) AND ((title:in) OR  
(description:in) OR (author:in)) AND ((title:action) OR (description:action)  
OR (author:action))
```

相比之下，eDisMax 查询解析器通过指定查询和查询字段 (qf)，更为轻松地实现对多个字段进行搜索：

```
q=solr in action&qf=title description author
```

这个示例查询构建起来更简单，它可以将内容分别放入多个字段中。eDisMax 解析器能更好地组织数据，数据不被挤在一个字段里，而且还可以帮助每个字段分

别进行 idf 统计, 改进相关度评分。保持字段分开的另一个好处是, 根据需要为每个字段赋予不同的权重:

```
q=solr in action&qf=title^1.5 description author^3
```

根据意愿可以在每个查询基础上调整权重。图 7.4 介绍了字段权重调整的搜索原理。



图 7.4 举例说明 eDisMax 查询解析器如何将词项分配到不同权重的多个字段上。字段的字号大小表示用于相关评分的字段的相对权重

在图 7.4 里, 每个关键词都在 `qf` 参数指定的每个字段上进行搜索。字段名称的字号大小表示字段调整后的相对相关度权重。除了调整字段权重外, 还可以根据词项位置的邻近程度对内容调整权重, 这与 Lucene 查询解析器一节介绍的词项邻近权重方法类似, 下一节会具体介绍。

#### 7.5.4 查询与短语的权重调整

eDisMax 查询解析器的一个重要功能是, 调整彼此邻近的词项的相关度。使用 Lucene 查询解析器的典型查询, 不管词项是否彼此邻近, 或是否视为一个短语, 所有词项的相关度都是相同的。eDisMax 查询解析器的另一个功能是, 对独立于用户主查询的函数进行任意地相关度调整。这些影响相关度的功能将在第 16 章进一步介绍。这里只是简单介绍一下每个相关度影响参数, 大致了解一下它们的作用。

##### pf (短语字段)、pf2 和 pf3 参数

`pf` 参数用于调整那些 `q` 参数中所有词项彼此非常靠近的文档得分。`pf` 参数与 `qf` 参数使用相同的格式, 获取字段列表及可选的相应权重。eDisMax 查询解析器尝试对 `q` 参数中所有词项进行短语查询, 如果能在任何短语字段中找到确切的短语, 则对匹配的文档调整相应的权重。

除了 `pf` 参数, eDisMax 查询解析器还支持 `pf2` 和 `pf3` 参数。这些参数功能与 `pf` 参数类似, 不过不需要 `q` 参数中所有词项, 它们将词项分解为二元 (`pf2`) 或三

元 (pf3), 只对包含少量词项的文档调整权重。在查询 Solr finds relevant documents 中, pf3 参数会对包含短语 "solr finds relevant" 或 "finds relevant documents" 的文档调整权重, 而 pf2 参数会对包含短语 "solr finds"、"find relevant" 或 "relevant documents" 的文档调整权重。

### ps (短语间隔)、ps2 和 ps3 参数

使用 pf 参数时, 你可能不希望查询中的所有词项作为一个精确的短语出现。使用 ps (短语间隔) 参数可以指定查询中的词项间隔位置界限, 以此在短语字段上判断匹配情况。

eDisMax 查询解析器还支持 ps2 和 ps3 参数, 允许为 pf2 和 pf3 修改短语间隔值。若未指定 ps2 和 ps3, 则它们的默认值是 ps 参数。

### qs (查询短语间隔) 参数

正如 ps 参数可以对短语字段 (pf 参数) 上的短语匹配定义间距 (编辑距离), qs 参数对用户在主查询 q 参数上明确指定短语的处理方法类似。将 qs 参数视为重新定义要匹配的确切内容, 可以将间距默认值 0 (词项彼此相邻) 修改为更高的数值。

### tie (决胜局) 参数

当查询的词项与文档的多个字段匹配时, tie 参数可以决定如何处理这种情况。为匹配到的每个字段的每个词项计算其相关度得分, 默认情况下, 每个文档中得分最高的字段用于该词项的相关度计算。这是析取的最大得分, 也是该查询解析器得名“析取最大值”原因所在。这与 Lucene 查询解析器形成鲜明对比, Lucene 查询解析器通常将每个字段的每个词项的相关度得分相加, 计算出每个文档的综合相关度得分。

tie 参数决定了最匹配的字段之外的其他字段的词项相关度得分有多少应该贡献给总体相关度得分。tie 参数的默认值是 0.0, 这表示其他字段不贡献权重。如果 tie 参数值为 1.0, 则表示所有字段为总体相关度得分贡献它们的全部权重, 就跟 Lucene 查询解析器中的做法一样了。在这种情况下, 相关度评分使用的是析取和, 而不是析取最大值。

### bq (提升查询) 参数

bq 参数接受查询字符串, 其包含在主查询 q 参数中, 用来影响相关度得分。它不会修改匹配到的文档数, 只修改文档返回的顺序。如果想为最近的文档提升相关度, 可以在请求中添加以下内容:

```
bq=date:[NOW/DAY-1YEAR TO NOW/DAY]
```



这将有效提升日期属于去年的所有文档的相关度得分。另外，还可以指定多个 bq 参数，在查询解析时针对不同子句分别进行提升。

### bf (提升函数) 参数

正如 bq 参数能够通过另一个查询来提升主查询的相关度一样，bf 参数能够通过函数查询来提升主查询的相关度。函数查询会在第 15 章详细介绍，这里仅举例说明如何使用函数来提升日期较新的文档的相关度。

```
recip(rord(date),1,1000,1000)
```

bf 参数接受 Solr 支持的所有函数及其权重值。

## 7.5.5 字段别名

有时需要在 Solr 中使用内部字段名，这些字段名并不适合显示给用户。对于动态字段尤其如此，动态字段名可能类似 title\_t\_en 的这样，但是我们希望在搜索中使用对用户更友好的语法，例如，title:"some title"。eDisMax 查询解析器为此提供了字段别名机制。

eDisMax 查询中的字段别名通过在请求中添加参数 f.{alias}.qf={realfield} 来实现。在上一个例子中，Solr 查询如下所示：

```
/select?defType=edismax&q=title:"some title"&f.title.qf=title_t_en
```

在本例中，查询在 title\_t\_en 字段上执行，接下来它会被查询中出现的 title 字段替换。字段别名参数在默认的 qf 参数后使用，这意味着，可以将一个别名分别以不同的权重对应到多个内部字段。在请求中添加任意数量的别名也是可以的。举例来说，有以下字段：

```
personFirstName, personLastName, itemName, companyName, cityName, stateName,
postalCodeName
```

你可以使用以下 Solr 请求为用户简化查询：

```
/select?defType=edismax&
  f.who.qf=personLastName^30 personFirstName^10&
  f.what.qf=itemName company^5&
  f.where.qf=city^10 state^20 country^35 postalCode^30&
  q=...
```

在这样的请求中，用户可以使用以下语法进行查询：

```
who:(trex grainger) what:(solr) where:(decatur, ga)
```

图 7.5 解释了该查询的关键词在 Solr 索引字段之上的三个字段别名 who、what 和 where 的对应情况。

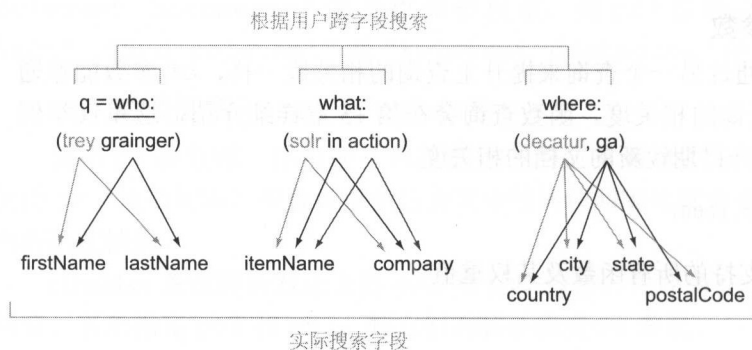


图 7.5 剖析了字段别名如何搜索对应的真实字段。这种机制可以为定义的任何虚拟字段有效地创建属于自己的 `qf` 参数，而不是仅限于定义虚拟默认字段

正如你所期望的，多别名字段的搜索类似于默认字段的搜索，其中每个查询项被分布在别名定义的查询字段上。唯一的区别在于，可以为每个别名定义单独的 `qf` 字段，而不是为默认字段定义一个 `qf` 参数。因此，与每个别名相关的查询部分会对字段列表进行搜索。

图 7.5 解释了词项在多个字段上的分布情况，但这不代表字段权重与每个底层的查询相关。内部指定的查询通过 `eDisMax` 查询解析器自动扩展为以下完整的查询，在 Solr 索引上进行搜索：

```

((
  (personFirstName:treY^10.0 | personLastName:treY^30.0)
  (personFirstName:grainger^10.0 | personLastName:grainger^30.0)
)
(
  (itemName:solr | company:solr^5.0)
)
(
  (state:decatur^20.0 | postalCode:decatur^30.0
   | country:decatur^35.0 | city:decatur^10.0)
)
(
  (state:ga^20.0 | postalCode:ga^30.0
   | country:ga^35.0 | city:ga^10.0)
)
)
  
```

将任意数量的字段别名映射为查询字段列表的做法，向用户提供了更简单的查询语法。这应该可以实现任意数量的字段权重规则，无须预先解析用户的查询或仅依赖于默认的 `qf` 参数。当然，如果不想向每个用户公开所有字段或别名，也是可以做到的。下一节介绍如何处理基于字段或别名的访问限制。

### 7.5.6 可访问字段

在许多情况下, 用户只能对默认字段以及 (可能的) 一小部分其他字段进行关键词搜索。由于有些内部字段可能会包含某些敏感信息 (例如, 用户 ID 或其他内部标识符), 你可能不希望用户从 Solr 索引中猜出其他字段并查询它们。

虽然 eDisMax 查询解析器允许主查询 `q` 参数对任何字段进行搜索, 但也可以使用 `uf` (用户字段) 参数加以限制。默认值是 `uf=*`, 允许用 `field:expression` 语法查询所有字段。如果要限制可用字段为单个 `title` 字段, 指定 `uf=title` 即可。多字段的访问使用空格隔开: `uf=title city date`。如果要对用户禁用所有字段, 则使用否定语法: `uf=-*`。如果要对指定字段列表之外的其他字段进行访问, 则使用 `uf=* -hiddenField1 -hiddenField2`。

为确保完全控制用户查询, 可以将 `uf` 参数与上一节介绍的字段别名参数结合使用。`uf` 参数既接受真实字段, 也接受别名, 因此构造出如下查询:

```
/select?defType=edismax&
  &df=text&
  f.who.qf=lastName^30 firstName^10&
  f.what.qf=itemName companyName^5&
  uf=who what&
  q=+who:(timothy potter) +what:(solr in action) +"big data"
```

在本例中, 查询扩展为: `timothy` 和 `potter` 在 `firstName` 和 `lastName` 字段上搜索, `solr`、`in` 和 `action` 在 `itemName` 和 `companyName` 字段上搜索。该查询还在默认的 `text` 字段上搜索短语 "big data"。如果查询尝试对 `who` 和 `what` 别名字段之外的其他字段进行搜索, 将会不起作用。以下面的查询为例:

```
q=+who:(timothy potter) +what:(solr in action) +firstName:timothy
```

在本例中, 查询不会将 `firstName` 解析为字段, 而是将默认的 `text` 字段中的完整短语作为关键词进行搜索。因此, 除非有文档包含 "firstName: timothy" 这个搜索文本, 否则该查询没有结果返回, 从而保护搜索引擎不访问未经授权的字段。如果需要在搜索应用中限制访问字段, eDisMax 查询解析器应该能处理此类情况。

### 7.5.7 最小匹配

在布尔逻辑讨论中已经介绍过二元运算符: `AND` 和 `OR`。它们是 Lucene 对必须匹配和应该匹配的内部表示形式。查询表达式 `hello AND world` 可以改写为 `+hello +world`, 这表示 `hello` 和 `world` 都必须匹配。查询 `big OR brown OR cow` 表示 `big`、`brown` 或 `cow` 其中一个词项必须匹配即可。然而, 如果一个查询要匹配多个表达式, 却又不介意匹配的是哪些表达式, 又该如何实现呢?

eDisMax 查询解析器通过 mm（最小匹配）参数模糊了传统布尔逻辑的界限。为了让文档实现匹配，mm 参数在查询中可以定义必须匹配的特定数量的词项或词项的百分比。这是对搜索应用的查准率与查询率进行操作的一个好工具。原因在于，它不要求所有词项必须匹配（默认运算符是 AND），或仅需要其中一个词项匹配即可（默认运算符是 OR）。

mm 参数语法很丰富，很难一下子掌握。表 7.2 给出了各种 mm 参数值及相关说明。

表 7.2 最小匹配语法示例

种类	最小匹配值	说明
正整数	2	2 个可选子句必须匹配。若少于 2 个，则所有子句都需要匹配
负整数	-3	除了 3 个可选子句之外，其他所有子句必须匹配
正百分比	75%	75% 的可选子句必须匹配。向下舍入到最接近的整数，因此小于 75% 是可能的
负百分比	-30%	至多缺失 30% 的可选子句。向下舍入到最接近的整数，因此不会超过 30% 的缺失
条件百分比	3<80%	如果有 3 个或更少的子句，则所有子句必须匹配。如果多于 3 个子句，则需要 80% 的子句匹配。这种情况下使用正百分比规则
多条件百分比	3<-1 5<4 7<-30%	如果有 3 个或更少的子句，则所有子句必须匹配。如果有 4 或 5 个子句，可以缺失 1 个子句。如果有 6 或 7 个子句，则 4 个子句必须匹配。如果多于 7 个子句，则至多 30% 的子句无须匹配

从表 7.2 可以看出，最小匹配语法规则内容非常丰富。它可以定义必须匹配的表达式数量（正整数）、遗漏的表达式数量（负整数）、必须匹配的表达式百分比（正百分数）以及遗漏的表达式百分比（负百分数）。要进一步控制的话，根据查询中现有的表达式数量，可以定义不同的最小匹配规则。

下面的查询结构为例：

```
/select?q={!edismax mm="2<50% 4<-45%" v=$example}&example=...
```

对于该查询，以下规则将对不同的示例参数值生效：

example=solr	必须匹配所有词项
example=solr is	必须匹配所有词项
example=solr is a	必须匹配 1 个词项（匹配率为 33%，四舍五入至 50%）
example=solr is a search	必须匹配 2 个词项（匹配率精确为 50%）
example=solr is a search engine	必须匹配 2 个词项（失误率为 40%，四舍五入至 45%）

eDisMax 查询解析器的最小匹配功能可以对包含多个关键词的查询进行匹配质量与数量的细粒度控制。图 7.6 从交集的角度解释了最小匹配阈值的作用。

`q={ledismax mm="40%"}solr is a search engine`

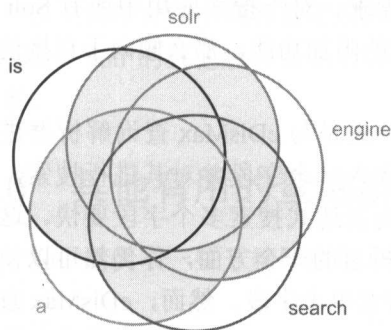


图 7.6 5 个词项的最小匹配值设置为 40%。在这种情况下, 5 个词项至少需要匹配 2 个。与第 3 章介绍的传统布尔逻辑 AND (所有词项) 与 OR (仅一个词项) 相比, 最小匹配支持不同的交集

在图 7.6 中, 阴影是词项所代表的圆的重合部分, 表示与该查询中词项相匹配的文档交集。你会注意到, 由于必须匹配 40% 的词项, 也就是 2/5, 这个文氏图看起来与第 3 章介绍的传统布尔运算符 AND 和 OR 有所不同。AND 运算符默认匹配所有的词项, OR 运算符默认仅匹配一个词项, 而通过 mm 参数可以实现更细致的匹配目的。mm 参数变大通常会提高查准率, mm 参数变小通常会提高查全率。有关查准率与查全率请参考第 3 章。

不过, 使用 mm 参数需要注意以下几点。首先, 如果计算中确定不需要子句, 例如, 指定的多个表达式可以不包含在查询中, 那么默认的布尔逻辑至少需要一个子句才能使用。这同样适用于上限。从本质上讲, 小于 1 的最小匹配值永远不会被使用 (因为至少需要一个), 大于子句数量的最小匹配值也不会被使用 (因为最大值是查询中的子句数量)。其次, 当处理百分比时, 正负百分比在边界情况上使用可以实现不同的作用。例如, 处理 5 个子句时, 80% 和 -20% 产生的结果相同; 处理 4 个子句时, 80% 表示需要 3 个子句, 即 80% 约等于 3/4, 不超过 4 个, 而 -20% 表示 4 个子句都需要, 即 -20% 约等于 0, 缺失的不超过 1 个。eDisMax 查询解析器的这些细微差别和整体表达能力使其成为查询工具包中的一个有力工具。

### 7.5.8 eDisMax 的优缺点

eDisMax 查询解析器除了支持 Lucene 查询解析器的所有查询语法之外, 还提供了许多附加功能。例如, 多字段搜索、清理用户输入、字段别名与字段限制, 以及通过多查询修正来改进短语相关度和其他权重因素。eDisMax 查询解析器包含了 Lucene 查询解析器的所有可用功能, 你可能想知道, 为什么有人还会考虑直接使

用 Lucene 查询解析器呢？

对于典型的面向用户的搜索应用，一般会使用 eDisMax 查询解析器。但是，在搜索应用程序层使用 eDisMax 查询解析器重新实现对用户友好的查询功能，这样做是无意义的，除非搜索应用中包含非常特殊的需求。对于搜索应用中所有 Solr 请求的生成，如果不需要 eDisMax 查询解析器提供的附加功能，那么倾向于直接使用 Lucene 查询解析器。

使用 eDisMax 查询解析器也有一些缺点。首先是与 eDisMax 查询解析器进行多字段搜索相关的处理问题。如果将所有词项放入一个字段并对其进行搜索，查询速度比使用 eDisMax 查询解析器在相同的查询表达式搜索多个字段要快。这不是 Lucene 查询解析器本身优于 eDisMax 查询解析器的一个方面，你仍然可以使用 eDisMax 查询解析器像 Lucene 查询解析器那样搜索单个字段。然而，eDisMax 查询解析器可以轻松实现多字段搜索的同时，会为许多基于 Solr 的搜索应用带来额外的执行花销。

eDisMax 查询解析器对相关度评分的影响应予以考虑。Lucene 查询解析器会考虑 `q` 参数中每个词项的相关度，不管该字段是否被搜索过。与之不同，eDisMax 查询解析器仅考虑与词项匹配（默认情况下）的得分最高字段的相关度。以下面的两个查询为例来说明：

```
/select?q={!edismax qf=title content}solr  
/select?q=title:solr OR content:solr
```

理论上，与这些查询匹配的文档的相关度得分相同，这是因为 eDisMax 查询要求 Solr 对两个字段都进行搜索。然而，从内部看，默认情况下 eDisMax 查询只使用与词项最匹配的字段的相关度。因此，如果 `solr` 的相关度在 `title` 字段更高，那么只有 `title` 字段的相关会被用于相关度评分，同样道理也适用于 `content` 字段。Lucene 查询解析器的处理方法不同，它会使用每个字段的得分来提高总体相关度，因此返回的相关度得分更加复杂。eDisMax 查询解析器的全称是扩展的析取最大（Extended Disjunction Maximum），字面意思也就是说，仅使用默认的析取得分的最大值。现实中某些字段比其他字段重要，eDisMax 查询解析器会对最佳字段中的文档与词项的匹配情况进行评分，而不对文档做整体评分。

客观讲，在大多数情况中使用这种方式会得到较好的相关度得分，不过你还是需要了解一下 eDisMax 查询解析器的相关度处理方式。eDisMax 查询解析器通过 `tie` 请求参数改变这种行为。如果指定 `tie=0`（默认值），就会得到相应的析取最大值。如果指定 `tie=1.0`，就会得到 Lucene 查询解析器的析取值之和。

另外，也可以将 `tie` 参数的取值介于 0.0 与 1.0 之间，若最匹配的字段拥有相关度权重的绝大多部分，则 `tie` 参数值越接近 0.0。大多数 Solr 应用对相关度并不

那么敏感，不会注意到评分间的这种细微差异。因此，对于大多数新建的搜索应用，推荐使用 **eDisMax** 查询解析器及其默认配置。如果相关度方面的需求非常关键，你需要关注所有这些细节，第 16 章会介绍更复杂的方法来提升搜索应用的相关度。

以上详细介绍了两种最常用的查询解析器：**Lucene** 查询解析器和 **eDisMax** 查询解析器，**Solr** 的其他解析器也有必要介绍一下，下一节简要介绍一些有趣的查询解析器。

## 7.6 其他有用的查询解析器

**Solr** 提供了一些开箱即用的查询解析器，本节简要介绍其中一些有趣的查询解析器。

### 7.6.1 字段查询解析器

字段查询解析器在指定字段中搜索词项或短语，可以使用该字段定义的任何文本分析方法。**f** 参数指明要进行词项或短语搜索的字段，其语法如下所示：

```
{!field f=myfield}hello world
```

该语法与使用 **Lucene** 查询解析器搜索短语 `myfield:"hello world"` 是等价的。

### 7.6.2 词项查询解析器和原始查询解析器

词项查询解析器可以直接在 **Solr** 索引上进行检索，但不能使用字段上定义的文本分析方法。这是它与字段查询解析器的不同之处。词项查询解析器可以对分面搜索（详见第 8 章）返回的值进行过滤，或对 **Solr** 索引中直接提取的词项组件进行过滤。词项查询解析器的语法如下：

```
{!term f=mystemmedtextfield}engin  
{!term f=mystringfield}Single Term with Spaces  
{!term f=myintfield}1.5
```

在之前的例子中，搜索到的值是查询提问中该字段在 **Solr** 索引中词项的可读版本。与字段查询解析器一样，**f** 参数指向的是搜索的字段。

**Solr** 还提供一个类似的实现形式——原始查询解析器。词项查询解析器与原始查询解析的唯一区别是，原始查询解析器在 **Solr** 索引中搜索确切的词项，而词项查询解析器搜索该词项的可读版本。

在特定字段中，例如，为了提高搜索效率，数值字段的内部存储采用了 **trie** 结构，此时词项查询解析器接受数值的可读版本（1.5），而原始查询解析器接



受 Solr 索引中该字段内部存储的机读版本。整数字段的数值 1 可能表示 Solr 索引中诸如 `#8;#0;#0;#0;#1` 此类的词项 trie 结构。以下两个查询都返回包含整数 1 的文档。

```
{!term f=myintfield}1  
{!raw f=myintfield}`#8;#0;#0;#0;#1;
```

你可能已经发现,原始查询解析器是一个高级功能,在典型的搜索应用中很少用到。Solr 的美好之处在于,用户不必完全理解那些让搜索更加有效的内部数据结构,就可以执行高效的搜索。因此,建议选择使用词项查询解析器,而不是原始查询解析器。

### 7.6.3 函数查询解析器和函数区间查询解析器

Solr 更强大的一个功能是在搜索过程中使用函数查询来生成动态值。这样的动态计算值包括确定地理空间距离、执行数学计算、转换字符串或在自定义的函数插件中执行任意代码。函数可能相当复杂,我们会用几乎一章的篇幅来详细介绍这一内容丰富的主题。函数查询解析器和函数区间查询解析器会在第 15 章介绍,到时还会介绍如何编写自定义的函数插件。

### 7.6.4 嵌套查询和嵌套查询解析器

截至目前,查询解析器是单独分别介绍的。我们已经了解如何为给定查询更改查询解析器,如果需要以特定方式将多个查询解析器组合起来,那该如何做呢?

Lucene 查询解析器和 eDisMax 查询解析器的查询语法支持一个特殊的运算符 `_query_`, 利用它可以轻松地对默认的 Lucene 查询解析器中其他查询解析器进行替换。这使得我们可以在任意复杂的布尔表达式中组合不同的查询解析器。

执行嵌套查询的语法为 `_query_:"[QUERY]"`, 其中 `[QUERY]` 表示在 `q` 或 `fq` 参数中单独使用的任意查询。以下面的查询为例说明:

```
/select?q=category:("technology" OR "business") AND  
_query_:"{!edismax qf=title^10 category^4 text}solr lucene hadoop mahout"
```

在本例中, eDisMax 查询嵌套在 Lucene 查询解析器的一个请求中。完整的嵌套查询必须使用引号括起来, 这表示查询内部的任何引号都必须使用反斜杠转义。应该注意的是, 在许多情况下没有必要显式使用 `_query_` 语法, 当 Lucene 查询解析器和 eDisMax 查询解析器发现查询中包含局部参数时, 一般会推测出需要使用嵌套查询。越是复杂的查询, 在 `_query_` 语法中需要使用嵌套查询的可能性就越大,

这样做是为了确保按查询意图进行解析。

除了特殊的 `_query_` 运算符之外, Solr 还提供了一个内置的嵌套查询解析器, 它也能处理嵌套查询。嵌套查询解析器的局部参数类型是 `query`, 通过以下方式调用:

```
/select?q={!query v=$nestedQuery}
```

以这种方式定义查询时, 可以用任意查询进行替换, 包括新的局部参数部分。如果要在 `solrconfig.xml` 中预先定义部分查询, 之后动态地替换查询解析器类型和查询值的话, 这种方式会非常有用:

```
<lst name="defaults">  
  <str name="nestedQuery">{!func}product(popularity, 0.25)  
</str>
```

在本例中, 一个函数查询替换了 `solrconfig.xml` 中的查询, 省去了创建查询时必须知道被替换的查询是什么类型这一步骤。当然, 嵌套查询解析器可以与 `_query_` 嵌套查询语法结合起来使用, 用来创建功能复杂的查询表达式, 甚至可能是嵌套和替换了多个层级的查询。现实中往往大多数搜索应用都不会那么复杂。

嵌套查询功能可以在主查询中将任意数量的子查询替换为不同的查询解析器, 支持所有的布尔表达式, 以确定如何组合这些查询。这一功能对通过函数提升查询的相关度非常有用, 包括地理空间距离和相关度得分的其他函数计算。

### 7.6.5 调整权重查询解析器

调整权重查询解析器允许根据一个文档是否与特定查询匹配, 自行定义相关度的调整策略, 无须过滤掉与要调整的查询不相匹配的文档。你可能还记得, `q` 参数通常用于过滤搜索结果, 并获得相关度评分中涉及的表达式的相似度情况。调整权重查询解析器可以提交相关度评分中所涉及的词项, 而不将其作为过滤器使用。调整权重查询解析器的语法如下所示:

```
{!boost b=1000}shouldboost:true  
{!boost b=log(popularity)}category:trending  
{!boost b=recip(ms(NOW,articledate),3.16e-11,1,1)}category:news
```

这三个例子对那些与指定查询值匹配的文档的相关度进行调整, 但不会将文档限制为仅与该查询值匹配的那些文档。另外, 也可以使用嵌套查询将调整权重查询解析器与其他查询解析器结合使用:

```
/select?q=_query_:"{!edismax qf=title content}data science" AND  
_query_:"{!boost b=log(popularity)}*:*" AND  
_query_:"{!boost b=recip(  
ms(NOW,articledate),3.16e-11,1,1)}category:news"
```

该查询会搜索关键词 data science，并根据流行度和发布时间（如果文档属于“新闻”类别）对这些文档进行相关度调整。该查询的结果数量与直接搜索 data science 的结果数量相同，对子句的权重调整只是影响了文档的相关度。

### 7.6.6 前缀查询解析器

前缀查询解析器可用于通配符查询，其语法如下所示：

```
{!prefix f=myfield}engin
```

该查询等同于使用 Lucene 查询解析器搜索 myfield:engin\*，对 myfield 字段匹配诸如 engine、engineer 或 engineering 这样的词项。应该注意到，前缀是直接对 Solr 索引进行搜索，所以在与索引比对之前，不会对输入进行文本分析。采用词项的索引表示是为了避免前缀输入与要搜索的词项之间出现匹配失误。在大多数情况下，更好的做法是使用 Lucene 查询解析器或 eDisMax 查询解析器进行通配符搜索。

### 7.6.7 空间查询解析器

Solr 提供丰富的地理空间搜索功能，可以在索引和查询阶段定义位置或形状。通过给定一个具体点位可以过滤出特定范围内的文档。例如，给定旧金山的经纬度可以过滤出 50 公里半径内的所有文档。Solr 提供两个用于空间查询的查询解析器，一个是空间框查询解析器 (bbox)，另一个是空间过滤查询解析器 (geofit)。空间搜索是 Solr 的一个重要主题，第 15 章会用大量篇幅介绍如何用好 Solr 的地理和空间搜索功能，空间查询解析器的有关内容也会在那里介绍。

### 7.6.8 连接查询解析器

Solr 的连接查询解析器可以执行子查询，实现文档集的伪连接。例如，要对查询进行限制，可以对不同的文档集执行子查询，并将原始查询结果集的限制条件设为文档只包含子查询文档中出现的字段值。Solr 的连接功能还可以用于跨 Solr 内核的文档。连接功能与函数查询功能、空间搜索功能一样，它们都属于高级功能，第 15 章会详细介绍，连接查询解析器也会在那里介绍。

### 7.6.9 分支查询解析器

分支查询解析器根据一些逻辑条件在多个查询/过滤器之间做出选择。它的操作与许多编程语言的分支语句类似。分支查询解析器的语法如下所示：

```
fq={!switch
  case.day='date:[NOW/DAY-1DAY TO *] '
  case.week='price:[NOW/DAY-7DAYS TO *] '
  case.month='date:[NOW/DAY-1MONTH TO *] '
  case.year='date:[NOW/DAY-1YEAR TO *] '
  case.else='*:*'
  v=$withinLast}
```

对于该过滤器，在 Solr 请求中可以传入以下参数，选择其中一条分支：  
withinLast=day, withinLast=week, withinLast=month, withinLast=year. case.else 表示其他值将默认搜索所有文档。

通过这个简单日期例子的启发，你可能会想出分支查询解析器的很多种用法。与分支语句匹配之后，可以选中任意值和执行查询。因此，通过分支查询解析器将业务规则映射到 Solr 查询，使用简单的请求参数轻松地调用该查询。

### 7.6.10 外围查询解析器

为了较为全面地介绍 Solr 自带的查询解析器，这里介绍一个适用范围很小的查询解析器——外围查询解析器。该查询解析器是为跨度查询的充分使用而设计的，其用于掌握词项彼此之间的位置关系。外围查询解析器使用特殊的运算符 *n*（有序）和 *w*（无序），前面带一个 1 ~ 99 之间的整数值。举例如下：

```
{!surround}3w(solr, action)
{!surround}5n(solr, action)
{!surround}solr 3w action
{!surround}solr 3n in 2w action
```

第一个例子在距离词项 *action* 3 个位置之内（无序、向前或向后）找到词项 *solr*。第二个例子在距离词项 *action* 5 个位置之内（*action* 必须位于 *solr* 之后）找到 *solr*。这两个例子都使用前缀表示，词项用圆括号括起来，运算符 *3w* 和 *5n* 作为函数，对输入值进行运算。

第三个例子与第一个例子在逻辑上是一样的，但它没有使用前缀表示，而是使用中缀表示。第四个例子展示了多个词项的组合，其中词项 *solr* 必须出现在距离词项 *n* 之后的三个位置之内，而词项 *action* 必须出现在距离词项 *in* 之前或之后的两个位置之内。

外围查询解析器还支持在前缀表示中使用 AND 和 OR 运算符，在中缀表示中使

用 AND、OR、NOT 以及 () 运算符。虽然增加的邻近运算符似乎是 Solr 可用查询语法的有力补充,但外围查询解析器存在一个主要限制:它不支持文本分析。遗憾的是,与词项解析器类似,外围查询解析器不先执行字段类型定义的分析器,而是直接在 Solr 索引上检索词项。虽然外围查询解析器支持常见查询运算符,例如,字段权重 (`field^10`) 和通配符 (`hel*w?rld`),但不支持文本分析会降低它对典型的关键词搜索场景的可用性。除非在构造查询时你对字段所需的文本分析有着深度了解,并能重构该查询。

### 7.6.11 最大得分查询解析器

当使用 Lucene 查询解析器对查询进行评分时,每个词项和子句都会被评分,所有这些评分汇总在一起作为每个文档的总相关度得分。在某些情况下,使用多个子句的最大得分会比使用子句的总得分更好一些。`MaxScoreQParserPlugin` 的使用方法如下:

```
{!maxscore}term1 term2 term3
```

在该查询中,每个文档都要对这三个词项进行评分,得分最高的词项(而不是三个词项的得分和)将被用作整体相关度得分。如果有需求,可以在同一个查询中结合使用两个查询解析器:

```
/select?q=one OR two OR _query_:"{!maxscore v=$maxQ}"&  
maxQ=three OR four OR five
```

与该查询匹配的所有文档的总得分由三部分组成:词项 `one` 的得分、词项 `two` 的得分与词项 `three`、`four`、`five` 中的最大得分。`MaxScoreQParserPlugin` 扩展了 `LuceneQParserPlugin`,因此,要对所有子句的得分求和或仅使用所有子句最大得分,选择其中一种查询解析器即可。

### 7.6.12 折叠查询解析器

折叠查询解析器能够从搜索结果集中删除重复的文档(即这些文档包含指定字段中的相同值),这称为字段折叠。在确保搜索结果多样性方面,它起到了重要作用。在 Solr 中,字段折叠有两种实现方法。第一种是使用功能更丰富的结果分组,将字段中具有相同值的文档合在一起,每一组仅返回文档的具体数量。第二种是使用折叠查询解析器,将相同字段值的文档合在一起。第 11 章会详细介绍结果分组和字段折叠,你可以在 11.7 节中找到折叠查询解析器的具体用法。

至此,你已经了解了 Solr 的很多查询解析器。除了 Solr 的标准查询解析器之外,你应该也能以插件方式编写自己的查询解析器了。对于大多数用户而言, Solr 的内

置查询解析器其实已经能满足所有需求了。对查询解析器构造查询的方法有了充分理解之后, 接下来介绍如何处理查询结果。

## 7.7 返回搜索结果

本章已经介绍了如何构造查询来找到所需的匹配文档集, 执行查询最终会返回一些结果。在许多情况下, 我们希望返回一小部分文档并显示在页面上, 每个文档包含一个或多个字段, 还要根据相关度对这些文档进行排序。在另一些情况下, 我们可能希望为每个字段返回动态计算值, 或指定返回的最大文档数和偏移值, 对完整的文档列表进行分页显示, 或根据其他因素(非相关度)对文档进行排序。本节介绍搜索结果的排序、返回所需字段和文档数, 以及为搜索应用选择适当的响应格式。

### 7.7.1 选择响应格式

为了便于阅读, 本书的绝大多数例子都选择 JSON 作为响应格式。之前你已经见过几个 XML 结果示例, 这是 Solr 的默认响应格式。从 Solr 的角度看, 什么样的响应格式并不重要。第 4 章介绍过, Solr 可以返回 XML、JSON、Ruby、Python、PHP、二进制 Java 等, 甚至还可以返回自定义格式。

使用 `wt(write type)` 参数修改响应格式。Solr 的 `wt` 参数的可用格式如表 7.3 所示。

表 7.3 Solr 可用的响应编写器

wt 值	相关的 Solr 类
csv	CSVResponseWriter
json	JSONResponseWriter
php	PHPResponseWriter
phps	PHPSerializedResponseWriter
python	PythonResponseWriter
ruby	RubyResponseWriter
xml	XMLResponseWriter
xslt	XSLTResponseWriter
javabin	BinaryResponseWriter

为了更改 Solr 的响应格式, 需要在请求中将 `wt` 参数设置为表 7.3 中的某些值。代码清单 7.4 使用三种可用格式表示搜索结果。

## 代码清单 7.4 不同的 Solr 响应格式举例

## 查询请求 1

```
/select?wt=json&indent=true&q=*&fl=id,title&rows=2
```

## 搜索结果 1

```
{
  "responseHeader":{
    "status":0,
    "QTime":1,
    "params":{...},
    "response":{"numFound":1000,"start":0,"docs":[
      {
        "id":"1",
        "title":"solr in action",
      },
      {
        "id":"2",
        "title":"lucene in action",
      }
    ]}
  }
}
```

## 查询请求 2

```
/select?wt=xml&indent=true&q=*&fl=id,title&rows=2
```

## 搜索结果 2

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      ...
    </lst>
  </lst>
  <result name="response" numFound="1000" start="0">
    <doc>
      <str name="id">1</str>
      <str name="title">solr in action</str>
    </doc>
    <doc>
      <str name="id">2</str>
      <str name="title">lucene in action</str>
    </doc>
  </result>
</response>
```

## 查询请求 3

```
/select?wt=csv&indent=true&q=*&fl=id,title&rows=2
```

## 搜索结果 3

```
id,title
1,solr in action
2,lucene in action
```



正如你所见,使用 `wt` 请求参数就可以轻松地配置 Solr 的响应格式。如有需要,还可以为搜索应用编写专门的响应格式。为此,你需要编写一个继承 Solr 的 `QueryResponseWriter` 的类。然后,在 `solrconfig.xml` 文件中注册该响应读写器,如下所示:

```
<queryResponseWriter name="myapp" class="...MyAppResponseWriter" />
```

由于大多数现代编程语言都拥有简单易用的代码库来处理 XML 和 JSON 格式,一般没必要花时间来编写一种响应格式,虽然这样做肯定是可以的。如果出于严格的安全考虑,例如,在返回给请求程序之前需要对结果进行加密,你可能会想到要自定义一个响应读写器。或者,需要对 Solr 返回的字段进行限制,下一节会讨论这个问题。

### 7.7.2 选择返回字段

在 Solr 的 `schema` 中指定字段时,其中有一个选项是指定该字段是被否存储。第 5 章创建索引时介绍过存储字段,它可以将原始文本与搜索结果一并返回。然而,并不是所有存储字段都必须在搜索结果中与文档一并返回。事实上,如果只需一小部分字段的话,可以通过限制搜索结果中要返回的字段数来加速搜索响应,这样做也会减少搜索结果的大小。

#### 返回存储字段

你可能还记得第 2 章介绍过,字段列表 (`fl`) 参数决定从 Solr 返回哪些字段。字段列表定义了搜索结果中每个文档需要返回的字段,字段之间以逗号分隔,使用语法如下所示:

```
/select?...&fl=id,name  
/select?...&fl=*
```

第一个例子定义了每个文档需要返回 `id` 字段和 `name` 字段的存储值。如果要返回所有的存储字段, Solr 提供了一个通配符选项 (第二个例子中的星号), 这样就不需要明确指定字段了。除了返回存储字段之外, Solr 还可以返回动态生成的值,作为文档的伪字段。

#### 返回动态值

除了存储字段之外,文档还包含一种可以了解该文档的有用信息,即相关度得分。通过请求特殊的得分字段可以返回每个文档的相关度得分:

```
/select?...&fl=*,score
```

这个例子返回了文档的所有存储字段以及一个动态生成的相关度得分字段。由于相关度得分不属于存储字段，因此不能使用通配符自动返回，必须在请求中明确指定。当然，没有必要为了获得相关度得分而获取所有字段，只需请求 `score` 字段即可。

相关度仅仅是文档返回的各种动态值中的一种。Solr 还有函数查询功能，它能计算出文档的各种有趣值。举例来说，返回一个数学函数的计算值作为一个伪字段：

```
/select?...&fl=id,sum(integerField,10)
```

这个例子对整数字段的值加 10，将其作为文档的一个附加字段，与 `id` 字段一并返回。函数查询的动态计算能力将在第 15 章中详细介绍。除了对字段值进行动态计算之外，还可以通过文档转换步骤，进一步返回文档的一些其他信息。

文档转换

有时在最终文档写入 Solr 响应结果之前，获取文档的一些额外信息也是有帮助的。这些额外信息可能包括相关度得分如何计算的可读解释、分布式搜索中文档所在的分片（有关分片与分布式搜索，参见第 12 章），甚至是 Lucene 内部的文档编号。Solr 使用文档变换器来返回此类信息。文档变换器的调用方法如下所示：

```
/select?...&fl=*,[explain],[shard]
```

该请求调用了两个特殊字段：`[explain]` 和 `[shard]`。这些用方括号括起来的字段调用文档转换器会获得每个文档的一些元信息。表 7.4 是 Solr 内置的常用文档转换器列表。

表 7.4 Solr 的常用文档转换器

示例	解释
[docid]	Lucene 内部文档 ID（一个整数）
[shard]	生成结果的分片 ID
[explain]	相关度计算的解释
[explain style=nl text html]	特定格式（样式）的相关度计算解释
[value v=? t=int double float date]	每个文档都相同的一个具体值

`[docid]` 字段通常只用于需要与 Lucene 基础索引层面进行交互的情况中，这对大多数 Solr 用户来说不常见。`[shard]` 字段主要用于在分布式搜索中查找文档所在的 Solr 服务器和内核。`[explain]` 字段（详见第 16 章）有助于理解文档相关

度得分的计算原理。[value] 字段用于返回文档的静态值。

除了 Solr 内置的文档变换器，还可以通过继承 `org.apache.solr.response.transform.DocTransformer` 类，以插件方式编写自己的文档变换器，从而对整个文档进行操作与变换。要在 Solr 中使用自定义的文档变换器，需要创建一个工厂类来包装文档变换器，该工厂类继承自 `org.apache.solr.response.transform.TransformerFactory`，使用时需要在 `solrconfig.xml` 中注册这个变换工厂类：

```
<transformer name="magic" class="magicTransformer" >
  <string name="yourSetting">abracadabra</string>
</transformer>
```

虽然文档变换器不是 Solr 最常用的功能，但它在搜索结果的文档返回之前，为文档操作（字段的添加、删除和编辑）提供了有用的扩展点。

### 返回字段的别名

除了通过动态生成值返回伪字段之外，在搜索结果的文档返回之前，Solr 还提供了一种字段重命名方法。字段别名在实际字段名之前出现，使用冒号分隔，在搜索结果中使用一个新字段来返回实际字段名（或动态生成值）的值，别名则作为新字段的名称，举例说明如下：

```
/select?...&fl=id,betterFieldName:actualFieldName
```

如果想要使用 `fieldname_t_is`（该文本字段既索引也存储）这样的动态字段来返回一个用户友好的字段名 `fieldname`，字段别名方法在这里就能发挥作用了。只需在字段列表中指定 `fieldname:fieldname_t_is` 就可以了。另外，可以对任何字段以任何名称重命名。如果请求的字段是函数查询，提供比函数语法（默认字段名）更有理解含义的字段名称会变得更加容易。

第 15 章会进一步介绍如何在文档中将函数作为字段使用。在返回文档时，特别是请求返回许多字段时，要确保一次不要检索太多文档，以保持 Solr 响应结果的大小合理。这通常要求一次返回较少数量的文档，根据需要在多个请求中对结果进行分页，下一节将介绍搜索结果页面。

### 7.7.3 搜索结果分页

一个查询可能会匹配出许多文档（数百个甚至数十亿个），但是搜索结果通常以一页一页的方式显示给用户，每页包含一定数量的搜索结果。虽然 Solr 能够以毫秒级别对数百万或数十亿文档进行高效地查询匹配，但它并没有对返回大量文档的处理方面进行优化。Solr 返回数十个或数百个文档不在话下，但从千位数量级开始，

请求速度和吞吐速度会大幅降低。Solr 的最佳实践是,首先只返回最相关的一页结果,然后允许用户继续翻阅后面的结果页面。这样做的话,Solr 的每个请求都只返回有限数量的文档,避免了某个查询请求为了一次获取太多数据,占用资源并影响其他查询的执行速度的情况出现。

Solr 的搜索结果分页需要用到第 2 章介绍过的两个请求参数: start 和 rows。start 参数表示搜索结果返回的起始位置,从 0 开始。rows 参数表示该请求返回的文档数。假设有 ID 编号从 1 ~ 100 000 的 100 000 个文档,代码清单 7.5 给出了 rows 参数和 start 参数的一些组合方式。

代码清单 7.5 搜索结果分页

#### 查询请求 1

```
/select?q=*&sort=id&fl=id&  
  rows=5&  
  start=0
```

#### 搜索结果 1

```
{...  
  "response": {"numFound": 100000, "start": 0, "docs": [  
    {  
      "id": "1"},  
    {  
      "id": "2"},  
    {  
      "id": "3"},  
    {  
      "id": "4"},  
    {  
      "id": "5"}  
  ]}  
}
```

#### 查询请求 2

```
/select?q=*&sort=id&fl=id&  
  rows=5&  
  start=5
```

#### 搜索结果 2

```
{...  
  "response": {"numFound": 100000, "start": 5, "docs": [  
    {  
      "id": "6"},  
    {  
      "id": "7"},  
    {  
      "id": "8"},  
    {  
      "id": "9"},  
    {  
      "id": "10"}  
  ]}
```

```
}}
```

#### 查询请求 3

```
/select?q=*&sort=id&fl=id&  
rows=2&  
start=50000
```

#### 搜索结果 3

```
{...  
  "response": {"numFound": 100000, "start": 50000, "docs": [  
    {  
      "id": "50001"},  
    {  
      "id": "50002"}  
  ]}  
}
```

#### 查询请求 4

```
/select?q=*&sort=id&fl=id&  
rows=10&  
start=99997
```

#### 搜索结果 4

```
{...  
  "response": {"numFound": 100000, "start": 99997, "docs": [  
    {  
      "id": "99998"},  
    {  
      "id": "99999"},  
    {  
      "id": "100000"}  
  ]}  
}
```

正如你所见，一次返回一页是非常直观的。要查看下一页结果，例如，查看页面 1 之后的页面 2，将 `rows` 参数值与当前的 `start` 参数值相加，作为新的 `start` 参数值，继而重新执行查询，如查询 1 和查询 2 所示。查询 3 用来说明，可以指定搜索结果范围内的任何数字作为结果返回的起始位置。查询 4 用来说明，最后一页的结果数量可能比其他页面要少。因此，如果 `start` 参数的起始位置高于匹配的文档数量，则不会返回任何结果。各个搜索引擎的结果分页原理大体相同。

在这 4 个查询中，搜索结果都是按顺序排列的。事实上，你可能已经注意到，代码清单 7.5 使用了 `sort` 参数。Solr 可以对很多东西进行排序，包括字段和动态计算值的组合。下一节介绍排序功能。

## 7.8 搜索结果排序

搜索得到的结果按一定的次序返回。默认按关键词相关度得分进行排序，除此之外，日期、词项、数字以及函数等都可以作为排序依据。本节介绍搜索结果的排序机制。

### 7.8.1 按字段排序

执行关键词搜索，搜索结果默认按照文档的相关度得分以降序方式（得分越高越靠前，得分越低越靠后）进行排序。对于相同得分的文档，根据搜索索引的 Lucene 内部文档编号以升序方式进行排序。如果没有相关度得分，则按 Lucene 内部文档编号进行排序。通过 `sort` 参数可以轻松地在搜索请求中修改默认排序：

```
sort=someField desc, someOtherField asc
sort=score desc, date desc
sort=date desc, popularity desc, score desc
```

正如你所见，字段排序的语法是由字段和排序方向组成的列表，字段与排序方向之间使用空格分隔，各组字段排序之间使用逗号分隔。排序字段必须在 `schema.xml` 中标记为 `indexed=true`，排序方向分 `asc`（升序）和 `desc`（降序）两种。

按字段排序非常直观，不过有几种情况需要说明一下。首先，你应该知道排序是基于索引词项的次序。举例来说，使用字符串字段索引 1、2、3、10、20、30，它们按照字典顺序的排序为 1、10、2、20、3、30。

此外，字段不同语言的排序也不尽相同，Solr 内置了针对特定语言排序的分词过滤器，有关信息参见 <http://wiki.apache.org/solr/UnicodeCollation>。如果字段使用这个分词过滤器，搜索结果的排序可能符合该语言规则，但与标准字符串字段的排序有所不同。

需要注意一点：由于排序使用了索引词项，如果修改了索引词项，而非字段的原始值，那么排序结果可能对用户而言没有意义。如果在 Solr 中进行词项替换，那么最终的索引值（而非发送到 Solr 的原始值）将是排序的键。

#### 对缺失值排序

还有一种特殊情况是，文档排序中 `sort` 字段缺少取值。由于 Solr 默认情况下不要求大多数字段都必备，因此很容易出现一个字段只有部分文档具备。在这种情况下，如果要对该字段进行排序，与查询相匹配但不包含该排序字段的文档应该出现在搜索结果列表中的前面还是后面呢？视情况而定，Solr 为我们提供了 `sortMissingLast` 和 `sortMissingFirst` 两个属性，用来选择适合的字段排序行为，这两个属性在 `schema.xml` 中的定义如下所示：

```
<fieldType name="string" class="solr.StrField"
  sortMissingLast="true"
  sortMissingFirst="false" />
```

如果 `sortMissingLast` 属性设置为 `true`，则不管排序方向如何，不包含该字段值的所有文档都会显示在搜索结果排序列表的末尾。如果

`sortMissingFirst` 属性设置为 `true`，则不管排序方向如何，不包含该字段值的所有文档都会显示在搜索结果排序列表的开头。`sortMissingLast` 和 `sortMissingFirst` 的默认设置都为 `false`。在默认设置下，缺少的文档会以升序显示在任何一种排序方式的开头，或以降序方式显示在任何一种排序方式的末尾。

### 排序的内存占用

关于排序的最后一点，排序是一个非常占用内存的过程。为了对文档进行排序，Solr 使用 Lucene 的字段缓存，在首次请求排序时将字段的所有唯一值加载到内存中，可能也存在其他原因导致缓存已经加载过了。这意味着，要对包含数百万个唯一词项的索引进行排序的话，每个排序字段都会消耗大量内存。首次请求字段排序时需要构建内存结构，这可能导致初始排序查询变慢。这里不是说不应该对文档进行排序，而是让你意识到，需要有足够的内存来执行排序。此外，你可能希望通过示例搜索预热 Solr 实例来实现缓存加载，有关内容参见第 4 章。如果能使用不同的数据建模，则有可能避免对包含数百万个唯一词项的字段进行排序。

## 7.8.2 按函数排序

除了按字段排序，还可以根据函数查询的计算值进行排序。举例来说，根据距离某一点的地理距离（使用 `geodist` 函数）或文档的流行度与新旧程度（对流行度字段和日期字段使用数学函数）进行排序。甚至还可以将字段排序与函数排序结合起来使用，产生有趣的排序效果。第 15 章会详细介绍函数查询，你可以在 15.1.4 节中找到如何进行函数排序的详细讲解。

## 7.8.3 模糊排序

排序通常被认为是一个全有或全无的操作。事实上，通过 Solr 的相关度计算可以实现所谓的模糊排序。如果将相关度计算视为多因素的综合体（这里的多因素通常指关键词搜索，不过也可以搜索其他内容），那么通过对其进行提升，可以区分出查询中每个因素的权重。

将一个查询元素的重要性相较于另一个查询元素提升数千倍，这是提升相关度得分的一种直接手段。这样做的话，首先会匹配第一个查询元素的文档，其次才会匹配第二个查询元素。如果将相关度得分之间的差异控制在一定范围内，则可以进行模糊排序。相关度得分的两部分通常一个高于另一个，有时候也会存在一些重合，导致得分较低的文档的相关度高于得分较低的文档。

模糊排序在大多数情况下更多的是一种相关度考虑，它有很多种处理方法。第 11 章会介绍如何根据单独的选择标准返回一定数量的文档，第 15 章和第 16 章会介



绍如何使用函数查询来影响文档的排序，以期最大限度地改进搜索应用的相关度。在许多情况下，应该根据相关度进行排序，构造相应的查询，对返回文档以期望的次序进行排序，而不是使用之前章节讨论的字段硬排序技术。虽然有时两者都可以排序，但应该视搜索应用需求而定。

## 7.9 调试查询结果

即使你拥有丰富的 Solr 使用经验，仍有可能对有些搜索结果感到困惑。例如，查询解析问题导致产生多种搜索结果、相关度得分有时不显示、查询执行时间变长等。所幸，搜索处理器附带的特殊搜索组件 DebugComponent 可以解决以上问题。

### 7.9.1 返回调试信息

在 Solr 请求期间，要了解查询背后发生了什么，最简单的方法是开启调试选项，在请求中传入 `debug=true` 参数，激活 DebugComponent，返回信息类似于代码清单 7.6 所示的信息。

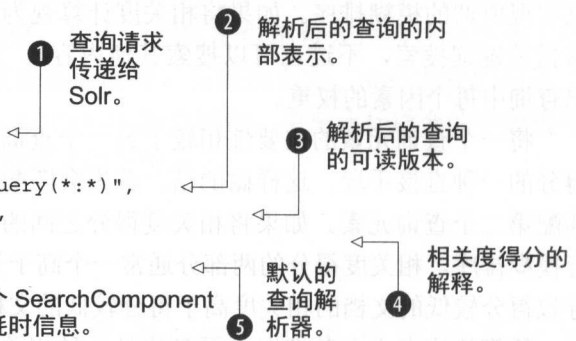
代码清单 7.6 请求调试信息

查询请求

```
/select?q=*&rows=3&debug=true
```

搜索结果

```
{
  "responseHeader": {...},
  "response": {... docs": [
    {
      "id": "1",
    },
    {
      "id": "2",
    },
    {
      "id": "3"
    }
  ] },
  "debug": {
    "rawquerystring": "*:*",
    "querystring": "*:*",
    "parsedquery": "MatchAllDocsQuery(*:*)",
    "parsedquery_toString": "*:*",
    "explain": { ... },
    "QParser": "LuceneQParser",
    "timing": { ... }
  }
}
```



从代码清单 7.6 中可以看出，DebugComponent 返回了一些有用信息：

❶ 原始查询 (rawquerystring), ❷ 解析后的查询在 Lucene 对象中的表示 (parsedquery), ❸ 解析后的查询的可读版本 (parsedquery\_toString), ❹ 文档相关度得分的数学计算过程 (explain), ❺ 查询中使用的默认查询解析器 (QParser), 以及 ❻ 每个 SearchComponent 处理所花费的时间 (timing)。

此外, 在调试参数中还可以进一步指定返回的具体内容。与指定 debug=true 不同, 如果指定 debug=query, 则只返回与查询解析器有关的部分调试信息。如果指定 debug=results, 则只返回相关度得分计算的解释内容。第 16 章会进一步介绍如何对相关度得分进行调试。如果指定 debug=timing, 则只返回每个 SearchComponent 处理所花费的时间, 这对调试执行较慢的请求很有帮助。通过调用 DebugComponent 的不同方面, 你可以更好地了解 Solr 请求的解析原理以及查询中哪些部分耗时最多, 这将有助于更有效地构建 Solr 请求。

## 7.10 本章小结

本章回顾了第 4 章中有关执行查询的请求处理流程。进行搜索时, SearchHandler 处理搜索请求, 调用一个或多个 SearchComponent。QueryComponent 是最重要的一种 SearchComponent, 负责执行主查询。本章讨论了 Solr 如何处理查询和过滤器, 介绍了 Solr 的各种查询器解析器及其语法。特别地, 在处理用户查询和多个字段搜索方面, 对比分析了 eDisMax 查询解析器与默认的 Lucene 查询解析器的优缺点。本章还介绍了如何提交查询、返回字段、对结果进行排序和分页、使用局部参数和解引用参数、使用 DebugComponent 调试查询问题等内容。本章还演示了在同一查询中如何结合使用多个查询解析器, 实现了在一个请求中使用全部查询功能。

至此, 你应该对执行查询、处理和调试搜索结果集有了较为充分的了解。返回搜索结果集的同时, 获取搜索结果集的有关汇总信息, 这将有助于更好地理解搜索结果并对其进行优化。Solr 的分面功能可以实现这一目的, 这是下一章要介绍的主题。

# 分面搜索

## 本章要点

- 将分面用于搜索结果的发现、分析与过滤
- 根据字段分面匹配相关文档，显示每个字段中排名最靠前的值
- 使用区间分面获得数值与时段的统计数
- 使用查询分面获得任意复杂查询的文档数
- 对搜索结果以外的值进行分面操作

与传统数据库及其他 NoSQL 数据存储相比，分面是 Solr 最强大的功能之一。分面搜索也称分面导航或分面浏览，它允许用户在执行搜索时，根据文档的一个或多个方面(即分面)对搜索结果进行细分。用户通过选择不同过滤器来探索搜索结果。

搜索新闻网站时，我们希望对搜索结果按照时间选项(前一小时、过去 24 小时、上一周)或类别选项(政治、技术、本地、商业)进行过滤。搜索求职网站时，我们希望对搜索结果按照城市、工作类别、行业甚至公司名称等选项进行筛选。一般来说，这些过滤选项不仅显示了每个分面的可用值，而且统计了每个分面值匹配到的搜索结果数。由于屏幕只能为每个分面显示有限数量的值，搜索引擎通常会根据热门程度(文档匹配最多的)对分面值进行排序。这让用户无须查看每个搜索结果，就可以快速了解搜索结果的整体情况。

Solr 的分面功能将动态元数据随搜索结果集一起返回。分面最基本的形式是展示字段中的每个唯一值，这个字段来自于许多文档共享的属性，例如，类目列表。

Solr 提供了许多高级分面功能,包括基于函数、值区间以及任意查询的搜索结果分面。Solr 还支持层级分面、多维分面与多选分面,其中多选分面可以对那些已经从搜索结果中过滤掉的文档进行分面计数。本章会介绍每一种分面用法,这些知识可以帮助你实现分面搜索体验。

为了充分理解本章内容,你应该熟悉如何执行查询和查询解析器的工作原理(参见第7章),还需要知道在 Solr 的 `schema.xml` 文件中,如何定义字段的文本分析方法(参见第5章和第6章)。了解 Lucene 索引如何存储词项对本章学习会有帮助,但这方面的知识不是必需的。

首先,我们通过示例来讲解如何在 Solr 中使用分面,快速掌握搜索结果整体情况,对搜索结果进行可视化。

## 8.1 搜索结果概览

本节概述分面方法,通过举例展示分面可视化的强大用途,以实现最大程度的可用性。分面搜索一般由两个步骤组成:分面的计算与显示,即“分面返回”,有时也简化为“分面”;用户选择一个或多个分面值,对搜索结果进行过滤,即“分面选择”或“分面过滤”。

基于这一点,你可能会想知道,搜索引擎返回的分面到底是什么?假设对一组餐馆文档进行搜索,你可能希望在用户界面中放置一些分面来作为导航元素,如图 8.1 所示,查询 `hamburger` 返回的分面。

► 餐馆类型	► 州	► 价格区间	► 城市
Fast Food (10073)	New York (4020)	< \$5 (5674)	New York, NY (2021)
Sit-down Chain (2530)	California (3459)	\$5 - \$10 (7000)	San Francisco, CA (1499)
Coffee Shop (1530)	Illinois (2450)	\$10 - \$20 (1007)	Chicago, IL (850)
Local Sit-down (998)	Georgia (1620)	\$20 - \$50 (1300)	Atlanta, GA (620)
Upscale (400)	Texas (1501)	\$50+ (550)	Austin, TX (501)
	...		...

图 8.1 搜索界面的导航元素展示了查询“hamburger”的分面搜索结果

导航元素对于分面是什么提供了清晰的视觉指示,揭开了分面强大功能的冰山一角。每个类别(餐厅类型、州、价格区间及城市)都被视为一个分面。分面可以看成是描述搜索结果的一个信息切片。在这个示例中,查询 `hamburger` 至少匹配到 14 000 个搜索结果,不过你能很容易地发现其中大多数是快餐店,多数餐馆位于大城市,而且多数餐馆的汉堡价格在 5 美元到 10 美元区间。

图 8.1 中提供的信息已经很有用了,但还有其他更好的分面可视化方法。显示所有提到的分面值会导致每个分面一次只能显示少数几个值。图 8.2 是州分面的另一种展示方式。

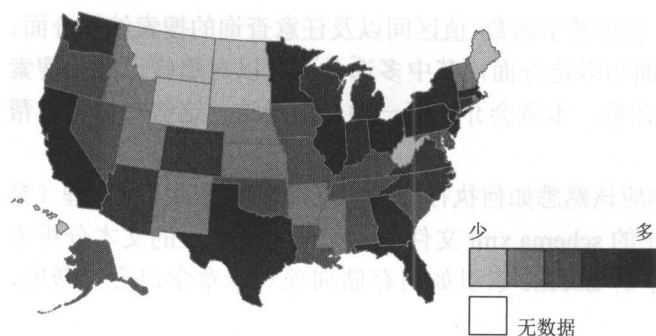


图 8.2 州分面的一种可视化呈现，将美国的 50 个州全部一次性显示在界面上，防止用户淹没在过量信息中

正如你所见，使用分面可视化来表示那些与用户搜索结果相关的重要元数据，这种方式可以增强搜索体验。图 8.3 是适合餐馆类型分面的一种可视化呈现。

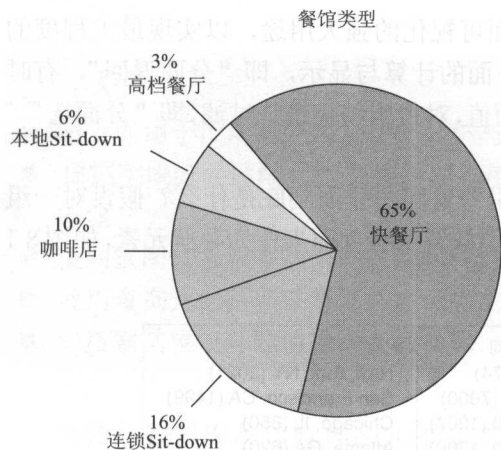


图 8.3 将餐馆类型分面以饼图形式可视化

地图和饼图可用于展示离散值，连续值(如数字和日期)一般用线图展示，如图 8.4 所示。

区间分面的线图非常有趣，这种可视化以不间断的序列方式绘制任意的值区间：数字、日期时间、价格、距离以及 Solr 内部的函数计算值。本章不是专门介绍数据可视化，这里就点到为止。不过，需要引出一个重要观点：分面提供了难以置信的强大功能，对用户搜索提供实时分析，有助于显著提升用户搜索体验。

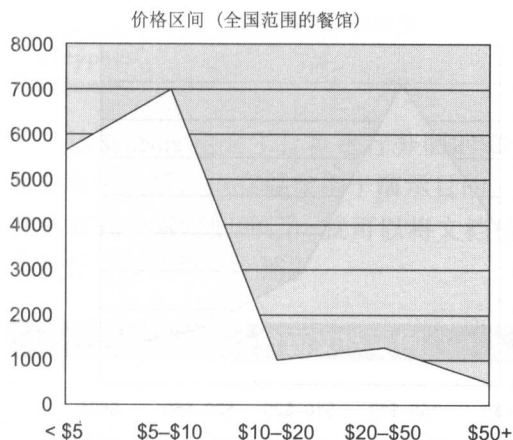


图 8.4 价格区间分面呈现为连续的线图，让用户对价格情况一目了然

在深入了解 Solr 的分面实现机制之前，还需注意一点，分面对每个搜索结果集（或从重复搜索的缓存中取回的结果）进行实时计算。每个分面返回一个分面值列表（例如，餐馆类型分面的快餐厅、西餐连锁、本地餐馆、高档餐厅），同时统计出每个分面值包含的文档数。由于一个分面值可能在同一个文档中多次出现，所以统计数值不是该分面值在所有文档中出现的次数，而是匹配到的文档数。事实上，所有分面值是根据搜索结果集进行计算的。这意味着，每当有新的搜索，每个分面返回的值与计数都不同。用户执行搜索，查看搜索结果中返回的分面，通过感兴趣的分面值对搜索结果集进行过滤，从而执行另一个搜索。

在图 8.1 中，如果用户点击州分面的 California 值会发生什么？搜索应用根据对该值的过滤，会执行另一个搜索吗？图 8.5 显示了第二次搜索之后的分面值。

<b>► 餐馆类型</b> Fast Food (1704) Sit-down Chain (799) Coffee shop (456) Local Sit-down (301) Upscale (199)	<b>► 州</b> California (3459)	<b>► 价格区间</b> < \$5 (800) \$5 - \$10 (1600) \$10 - \$20 (580) \$20 - \$50 (298) \$50+ (181)	<b>► 城市</b> San Francisco, CA (1499) Los Angeles, CA (701) San Diego, CA (535) San Jose, CA (356) Sacramento, CA (178) ...
---	---------------------------------	--	--

图 8.5 在图 8.1 的搜索请求之后，将州分面限定为加州，分面值的变化情况。请注意，城市分面中的所有城市现在都位于加州，搜索到的餐馆也都应该处于加州境内

图 8.5 只在单一值 (California) 上进行了过滤，还可以根据需要在任何给定搜索上使用多个过滤器，每个分面会根据使用的过滤器进行计算。细看第一个搜索（全国范围）的价格区间分面，与第二个搜索（California）的结果比较一下，你会发现加州餐馆与美国其他地方餐馆的价格差异，如图 8.6 所示。

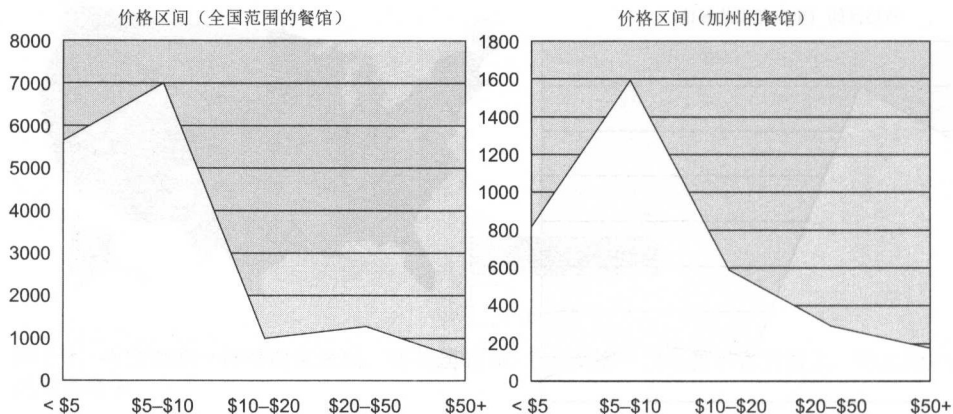


图 8.6 价格区间（全国范围）与价格区间（加州）的对比。从中可以显而易见地判断出，加州的餐馆价格总体上比美国其他地方的餐馆价格高

8.5 小节会介绍如何在分面值上使用这些过滤器。这里强调一个重要观点：分面为用户在查看特定查询的结果时提供了丰富的洞察力，用户能够很容易地评价搜索质量，再选择他们感兴趣的方面深入下去。

## 8.2 建立测试数据

在深入了解分面机制之前，我们需要在 Solr 中加载一些示例数据，以便于本章后续内容的示例讲解。示例数据是 8.1 小节餐馆示例文档的子集。本节创建并加载餐馆文档，供本章后续内容介绍中使用分面功能。向 Solr 提交这些示例文档，首先需要将代码清单 8.1 中定义的字段添加到 Solr 的 schema.xml 文件（参见第 5 章）。

代码清单 8.1 schema.xml 定义了示例餐馆文档的字段

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema name="example" version="1.5">
  <fields>
    <field name="id" type="string" indexed="true" stored="true" />
    <field name="name" type="string" indexed="true" stored="true" />
    <field name="city" type="string" indexed="true" stored="true" />
    <field name="type" type="string" indexed="true" stored="true" />
    <field name="state" type="string" indexed="true" stored="true" />
    <field name="tags" type="string" indexed="true" stored="true"
      multiValued="true" />
    <field name="price" type="double" indexed="true" stored="true" />
  </fields>
  <uniqueKey>id</uniqueKey>
  <types>
    <fieldType name="string" class="solr.StrField" sortMissingLast="true" />
    <fieldType name="boolean" class="solr.BoolField" sortMissingLast="true"/>
    <fieldType name="double" class="solr.TrieDoubleField" precisionStep="0" />
  </types>
</schema>
```

6 个字段。

应拥有唯一 ID。

3 个字段类型。



```

    positionIncrementGap="0"/>
  </types>
</schema>

```

该 schema 定义了打算进行分面的每个字段。每个字段被定义为既是索引字段也是存储字段，但存储仅用于演示目的。分面仅使用字段的索引值，如果不需要以其他方式检索它们的话，就可以将文档标记为仅索引而不存储。代码清单 8.2 是即将进行分面的每个文档。

### 代码清单 8.2 分面文档示例

```

[
  {
    "id": "1", "name": "Red Lobster", "city": "San Francisco, CA", "type": "Sit-
down Chain", "state": "California", "tags": ["sea food", "sit down"],
    "price": 33.00
  },
  {
    "id": "2", "name": "Red Lobster", "city": "Atlanta, GA", "type": "Sit-down
Chain", "state": "Georgia", "tags": ["sea food", "sit-down"],
    "price": 22.00
  },
  {
    "id": "3", "name": "Red Lobster", "city": "New York, NY", "type": "Sit-down
Chain", "state": "New York", "tags": ["sea food", "sit-down"],
    "price": 29.00
  },
  {
    "id": "4", "name": "McDonalds", "city": "San Francisco, CA", "type": "Fast
Food", "state": "California", "tags": ["fast food", "hamburgers",
    "coffee", "wi-fi", "breakfast"], "price": 9.00
  },
  {
    "id": "5", "name": "McDonalds", "city": "Atlanta, GA", "type": "Fast Food",
    "state": "Georgia", "tags": ["fast food", "hamburgers", "coffee", "wi-fi",
    "breakfast"], "price": 4.00
  },
  {
    "id": "6", "name": "McDonalds", "city": "New York, NY", "type": "Fast Food",
    "state": "New York", "tags": ["fast food", "hamburgers", "coffee", "wi-
fi", "breakfast"], "price": 4.00
  },
  {
    "id": "7", "name": "McDonalds", "city": "Chicago, IL", "type": "Fast Food",
    "state": "Illinois", "tags": ["fast food", "hamburgers", "coffee", "wi-
fi", "breakfast"], "price": 4.00
  },
  {
    "id": "8", "name": "McDonalds", "city": "Austin, TX", "type": "Fast Food",
    "state": "Texas", "tags": ["fast food", "hamburgers", "coffee", "wi-fi",
    "breakfast"], "price": 4.00
  },
]

```

每个文档都有 7 个字段。

每个餐馆的 id 字段是唯一的。

name 字段包含一个非唯一值。

city 字段包含一个非唯一值。

state 字段包含一个非唯一值。

tags 字段包含一个非唯一值。

price 字段是一个正数。

```

    "id":"9", "name":"Pizza Hut", "city":"Atlanta, GA", "type":"Sit-down
    Chain", "state":"Georgia", "tags":["pizza", "sit-down", "delivery"],
    "price":15.00
  },
  {
    "id":"10", "name":"Pizza Hut", "city":"New York, NY", "type":"Sit-down
    Chain", "state":"New York", "tags":["pizza", "sit-down", "delivery"],
    "price":24.00
  },
  {
    "id":"11", "name":"Pizza Hut", "city":"Austin, TX", "type":"Sit-down
    Chain", "state":"Texas", "tags":["pizza", "sit-down", "delivery"],
    "price":18.00
  },
  {
    "id":"12", "name":"Freddy's Pizza Shop", "city":"Los Angeles, CA",
    "type":"Local Sit-down", "state":"California", "tags":["pizza", "pasta",
    "sit-down"], "price":25.00},
  {
    "id":"13", "name":"The Iberian Pig", "city":"Atlanta, GA",
    "type":"Upscale", "state":"Georgia", "tags":["spanish", "tapas", "sit-
    down", "upscale"], "price":45.00
  },
  {
    "id":"14", "name":"Sprig", "city":"Atlanta, GA", "type":"Local Sit-
    down", "state":"Georgia", "tags":["sit-down", "gluten-free", "southern
    cuisine"], "price":15.00
  },
  {
    "id":"15", "name":"Starbucks", "city":"San Francisco, CA",
    "type":"Coffee Shop", "state":"California", "tags":["coffee",
    "breakfast"], "price":7.50
  },
  {
    "id":"16", "name":"Starbucks", "city":"Atlanta, GA", "type":"Coffee
    Shop", "state":"Georgia", "tags":["coffee", "breakfast"], "price":4.00
  },
  {
    "id":"17", "name":"Starbucks", "city":"New York, NY", "type":"Coffee
    Shop", "state":"New York", "tags":["coffee", "breakfast"], "price":6.50
  },
  {
    "id":"18", "name":"Starbucks", "city":"Chicago, IL", "type":"Coffee
    Shop", "state":"Illinois", "tags":["coffee", "breakfast"], "price":6.00
  },
  {
    "id":"19", "name":"Starbucks", "city":"Austin, TX", "type":"Coffee
    Shop", "state":"Texas", "tags":["coffee", "breakfast"], "price":5.00
  },
  {
    "id":"20", "name":"Starbucks", "city":"Greenville, SC", "type":"Coffee
    Shop", "state":"South Carolina", "tags":["coffee", "breakfast"],
    "price":3.00
  }
]

```

代码清单 8.1 的示例 `schema.xml` 和代码清单 8.2 的示例文档 `restaurants.json` 文件可以从本书官方网站下载。下载源代码并解压缩到一个文件夹, 书中使用 `$SOLR_IN_ACTION` 代替具体路径。执行以下命令来配置餐馆文档的 `schema`(代码清单 8.1) 并启动 Solr:

```
cd $SOLR_INSTALL/example/  
cp -r $SOLR_IN_ACTION/example-docs/ch8/cores/restaurants/ solr/restaurants/  
java -jar start.jar
```

Solr 成功运行之后, 使用以下命令向 Solr 提交代码清单 8.2 的文档:

```
cd $SOLR_IN_ACTION/example-docs/  
java -Durl=http://localhost:8983/solr/restaurants/update  
  -Dtype=application/json  
  -jar post.jar ch8/documents/restaurants.json
```

命令的最后一行使用 Solr 内置的 `post.jar` 工具, 从文本文件中索引了餐馆数据。这里指定的是 JSON 格式, 而不是默认的 XML 格式。如果一切顺利, 你应该会看到类似如下输出:

```
SimplePostTool version 1.5  
Posting files to base url  
http://localhost:8983/solr/restaurants/update using content-type  
=> application/json..  
POSTing file restaurants.json  
1 files indexed.  
COMMITting Solr index changes to  
=> http://localhost:8983/solr/restaurants/update..  
Time spent: 0:00:00.102
```

为餐馆文档成功建立索引后, Solr 的标准搜索处理器就可以发挥作用了。使用 `http://localhost:8983/solr/restaurants/select?q=*:*` 来确认 Solr 中索引的这些文档, 以及来自代码清单 8.2 的所有字段。

### 更改 Solr 的响应格式

请注意, 本章及后续章节中大多数文档和 Solr 的响应都采用 JSON 格式, 而不是 XML 格式。虽然 XML 是 Solr 的默认响应格式, 但 JSON 是一种易读的、更紧凑的格式, 更适合演示用途。将 `wt=json` 参数添加到 Solr 请求中, 将响应格式从 XML 变成 JSON 格式。如果没有在 `solrconfig.xml` 中将默认响应格式从 XML 修改为 JSON, 那么本章中 Solr 的所有请求都要需要添加 `wt=json` 参数, 以便以同一种格式返回结果。

你还会注意到，本章中所有 Solr 响应格式都经过缩进处理。在 Solr 的 URL 中添加 `indent=on` 参数来实现缩进。一般不推荐在生产环境中对响应格式进行缩进。虽然这样做是可以的，但会增加不必要的需要另外处理的空格。缩进会让 Solr 的响应更加可读，所以本章的所有例子使用缩进排版。

为避免重复说明，本章假设对所有查询都设置 `&wt=json&indent=on` 作为默认参数（具体做法参见第 4 章）。

Solr 给出的响应包含 20 个示例餐馆文档，每个文档包含 7 个已定义的字段，示例餐馆数据集如下所示：

```
{
  "responseHeader":{
    "status":0,
    "QTime":0,
    ...},
  "response":{"numFound":20,"start":0,"docs":[
    {
      "id":"1",
      "name":"Red Lobster",
      "city":"San Francisco, CA",
      "type":"Sit-down Chain",
      "state":"California",
      "tags":["sea food", "sit down"],
      "price":33.0,
      ...},
    },
    ...
  ]}
}
```

现在 20 个示例文档可以被搜索了，接下来开始执行分面搜索。我们从分面的最常见形式开始，即基于字段内的每个唯一值进行分面。

## 8.3 字段分面

字段分面是最常见的分面形式。执行搜索时，根据查询请求返回特定字段中找到的唯一值以及找到的文档数。8.1 节以可视化方式展示了餐馆例子用到的几个字段：餐馆类型（type）分面、州（state）分面与城市（city）分面。本节会介绍如何构建一个 Solr 查询来请求字段分面，如何调整分面参数来改变分面值的计算方式与返回方式。

出于本章其余内容的演示需要，在 8.2 节中索引过的文档上进行分面。让我们来执行第一个分面：

```
http://localhost:8983/solr/restaurants/select?q=*&rows=0&
facet=true&facet.field=name
```

查询结果详见代码清单 8.3。

### 代码清单 8.3 单值字段分面结果

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 43,
    "response": { "numFound": 20, "start": 0, "docs": [] }
  },
  "facet_counts": {
    "facet_queries": {},
    "facet_fields": {
      "name": [
        "Starbucks", 6,
        "McDonalds", 5,
        "Pizza Hut", 3,
        "Red Lobster", 3,
        "Freddy's Pizza Shop", 1,
        "Sprig", 1,
        "The Iberian Pig", 1
      ],
      "facet_dates": {},
      "facet_ranges": {}
    }
  }
}
```

所有分面的根节点。

列出所有字段分面。

我们在 name 字段请求一个分面。

在 6 个文档中找到 Starbucks。

在 1 个文档中找到 Sprig。

这个例子展示了 Solr 分面的基础形式——单值字段分面。在这种分面中，每个唯一值都会被检视，并返回每个值对应找到的文档数。由于每个文档只需要一个值，因此值的统计总数等于文档总数，这里的值是餐馆名称。

然而，并不是 Solr 的所有字段都包含单一值。标签 (tags) 字段就是多值字段的一个例子。如果对标签字段进行分面会怎样？详见代码清单 8.4。

### 代码清单 8.4 多值字段的分面

查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&facet.field=tags
```

搜索结果

```
...
"facet_fields": {
  "tags": [
    "breakfast", 11,
    "coffee", 11,
    "sit-down", 8,
    "fast food", 5,

```

分面的字段名称。

由于文档匹配多个标签，增至 20 以上。

```

    "hamburgers", 5,
    "wi-fi", 5,
    "pizza", 4,
    "delivery", 3,
    "sea food", 3,
    "gluten-free", 1,
    "pasta", 1,
    "sit down", 1,
    "southern cuisine", 1,
    "spanish", 1,
    "tapas", 1,
    "upscale", 1]
}
...

```

有趣的是, breakfast 与 coffe 是整个餐馆文档语料库中最流行的两个标签。这是因为星巴克和麦当劳这两家都被打上了这两个标签。如果搜索 breakfast 或 coffe, Solr 返回的结果与事实是吻合的。

还需注意一点, 标签分面值的总和远大于 20, 这个数字是搜索引擎中的文档总数。之所以出现这种情况是因为, 文档包含多个标签, 也就是说, 多个标签可以对应到同一个文档, 因此大多数文档被不止一个分面值计过数。

8.1 节曾讨论过分面可视化的几种方法。标签分面本身就可视为一种可视化类型——标签云。图 8.7 展示了基于标签字段的多值字段分面结果。



图 8.7 tags 多值字段的字段分面以标签云方式呈现。文字的大小对应该词项在文档中出现的次数。这再一次体现了对分面进行可视化的多种呈现形式

标签云是用户从分类角度概览搜索结果的一种常见方式。除了在这个例子中使用标签, 分面也常用于包含日常用语的原生内容字段(如餐馆描述全文), 以帮助用户快速了解内容字段的含义。对原生内容字段使用分面的问题是, 日常用语包含无意义词汇(诸如 and、of 和 like 这类词), 这样会产生很多噪声。在对文档的描述上, 原生内容字段就不如打标签那么可靠有效了。

另外还需注意, 进行分面时, 字段分面的返回值是基于该字段的索引值。这意味着, 如果传入词项 San Francisco, CA, 该字段会被作为文本字段进行分词, 以空格和逗号分隔, 并且文本会进行小写转换。然后, 在 Solr 索引中, 该字段的值

会变为 `ca`、`francisco` 和 `san`。因此，除非要对每个词项单独进行分面统计和小写转换，否则在 `schema.xml` 中创建字段定义时就要考虑字段的分面问题。一种标准做法是，Solr 开发者创建了一个单独的字段，在该字段中放入特定内容的副本，该字段仅用于分面目的，这样原始文本就能够以可分面的形式保留。

读到此处，你应该熟悉了分面的基本知识，也知道了如何在单值字段和多值字段上进行分面。至此，本章只介绍了返回了一个分面的默认设置。处理 20 个文档对分面来说很容易，如果在 Solr 的分面请求中需要返回数以千计或百万数量级的唯一词项，那又会怎样？所幸，Solr 拥有许多分面选项，允许对每个查询返回的分面进行微调。表 8.1 是字段分面的参数一览表。

表 8.1 在 Solr 的 URL 中通过指定字段分面参数来修改分面行为

Solr 参数	可能取值	说明
<code>facet</code>	<code>true</code> , <code>false</code>	对当前搜索启用或禁用所有分面
<code>facet.field</code>	任意索引字段的名称	确定对哪个字段进行分面计算。该参数可以多次指定以返回多个分面
<code>facet.sort</code>	<code>index</code> , <code>count</code>	根据最大出现次数 ( <code>count</code> ) 或索引 ( <code>index</code> ) 的词典顺序对分面值排序。该参数基于字段进行指定
<code>facet.limit</code>	整数 $\geq -1$	确定每个分面返回多少个唯一分面值。该参数基于字段进行指定
<code>facet.mincount</code>	整数 $\geq 0$	在分面值返回之前设置必须出现的最小文档数。默认情况下，当前搜索中没有匹配结果的词项会包含在分面结果中。因此，通常将 <code>facet.mincount</code> 至少设置为 1。该参数基于字段进行指定
<code>facet.method</code>	<code>enum</code> , <code>fc</code> , <code>fcs</code>	<code>enum</code> 方法循环遍历索引中的所有词项，计算这些词项与查询的交集。 <code>fc</code> (字段缓存) 方法对与查询匹配的文档进行循环，在这些文档中找到词项。对于包含许多唯一值的字段， <code>fc</code> 方法较快。对于包含值较少的字段， <code>enum</code> 方法较快。 <code>fc</code> 方法是除了布尔字段之外所有字段的默认方法。 <code>fcs</code> 方法为单值字符串调用每个片段的字段缓存。如果索引处在不断变化中， <code>fcs</code> 方法效果更好。 <code>fcs</code> 还接受局部参数 <code>threads</code> ，通过指定可用的线程处理不同的索引片段来加速分面进程。该参数基于字段进行指定
<code>facet.enum.cache.minDf</code>	整数 $\geq 0$	高级：在 <code>filterCache</code> 用于词项之前，指定与该词项匹配的最小文档数。默认值是 0，这表示 <code>filterCache</code> 应经常使用。设置该值大于 0，能减少较慢查询所消耗的内存。该参数基于字段进行指定
<code>facet.prefix</code>	任意字符串	指定字符串开头的词项来限制分面值。在查找相似词项时非常有用，常用于实现自动补全功能
<code>facet.missing</code>	<code>true</code> , <code>false</code>	指定是否在分面字段中返回所有不包含值 (值为缺失) 的文档计数



续表

Solr 参数	可能取值	说明
facet.offset	整数 >= 0	通过分面值进行分页。offset 指定跳过最开始的几个值，作为后面分面值的替代
facet.threads	整数	当执行多个字段分面时指定处理线程数。默认值是 0，所有字段分面依次执行。负数（“无限制”）为每个字段分面生成一个线程，正数创建 threads 指定的线程数。如果每次搜索执行多个字段分面，这会大大提升分面速度

表 8.1 中需要注意一点，通过指定多个 facet.field 参数可以实现多个分面的请求。此外，表中列出的一些分面参数应用在字段级别上，使用以下语法进行指定：

```
f.<fieldName>.<FacetParameter>=<value>
```

如果要对美国所有 50 个州(按字母顺序排列)进行分面，每个州至少有一个餐馆，而且要显示餐馆名（即使餐馆名与查询并不匹配）以及匹配到的前 5 个标签，执行代码清单 8.5 的查询。

代码清单 8.5 基于各个字段混合字段分面参数

查询请求

```
http://localhost:8983/solr/restaurants/select?q=*:*&
facet=true&
facet.mincount=1&
facet.field=state&
f.state.facet.limit=50&
f.state.facet.sort=index&
facet.field=name&
f.name.facet.mincount=1&
facet.field=tags&
f.tags.facet.limit=5
```

除覆盖的情况外，这是所有分面的默认值。

仅修改 state 分面。

仅修改 name 分面。

仅修改 tags 分面。

搜索结果

```
...
"facet_fields":{
  "state":[
    "California",4,
    "Georgia",6,
    "Illinois",2,
    "New York",4,
    "South Carolina",1,
    "Texas",3],
  "name":[
    "Starbucks",6,
    "McDonalds",5,
    "Pizza Hut",3,
    "Red Lobster",3,
```

```

    "Freddy's Pizza Shop",1,
    "Sprig",1,
    "The Iberian Pig",1],
    "tags":[
        "breakfast",11,
        "coffee",11,
        "sit-down",8,
        "fast food",5,
        "hamburgers",5]
}

```

正如你所见，代码清单 8.5 在多个字段上设置了分面选项，自定义了搜索结果返回的具体内容。虽然该查询需要设置许多参数，但分面选项带来的灵活性，值得去做一些额外操作。

至此，你已经学会了如何在 Solr 请求返回字段分面，在已索引的字段上查看与每个唯一值匹配的文档数。字段分面功能已经很强大了，但还有更加灵活的分面选项可供选用。接下来介绍查询分面，对无论多么复杂的查询都能返回分面计数。

## 8.4 查询分面

之前小节讨论过，对任意索引字段使用分面返回最靠前的值，这种做法很好。在任意子查询上使用这种做法也很有用，这样就可以知道未来的搜索可能匹配出多少结果，并基于该数字提供相应的分析。Solr 通过查询分面实现这个功能。

展示查询分面功能的最好方式是举例说明。参见 8.2 节的餐馆搜索数据集，假设我们要查询价格区间在 5 美元～25 美元的餐馆，而且想要了解美国东海岸、西海岸和中部地区的餐馆分布情况。通过 3 个不同的查询就可以达到查询目的，详见代码清单 8.6。

### 代码清单 8.6 执行多个查询，获取文档数，用于子查询

```
http://localhost:8983/solr/restaurants/select?q=*&fq=price:[5 TO 25]
```

```
...
```

```
"response":{"numFound":11 ...
```

```
http://localhost:8983/solr/restaurants/select?q=*&fq=price:[5 TO 25]&
fq=state:("New York" OR "Georgia" OR "South Carolina")
```

```
...
"response":{"numFound":5 ...
http://localhost:8983/solr/restaurants/select?q=*&fq=price:[5 TO 25]&
  fq=state:("Illinois" OR "Texas")
...
"response":{"numFound":3 ...
http://localhost:8983/solr/restaurants/select?q=*&fq=price:[5 TO 25]&
  fq=state:("California")
...
"response":{"numFound":3 ...
```

上面的示例告诉我们，Solr 子查询对搜索结果计数是最简单粗暴的方法，分别运行每个子查询就可以确定搜索结果总数。在本例的小数据集上，虽然这种方式没有操作不便之处，但这样做完全没有必要。代码清单 8.7 演示了如何使用查询分面轻松地将多个子查询组合起来。

#### 代码清单 8.7 执行单个查询分面，以获取文档数，用于子查询

##### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&fq=price:[5 TO
25]&facet=true&
facet.query=state:("New York" OR "Georgia" OR "South Carolina")&
facet.query=state:("Illinois" OR "Texas")&
facet.query=state:("California")
```

##### 搜索结果

```
...
"response":{"numFound":11,"start":0,"docs":[]},
  "facet_counts":{"
    "facet_queries":{"
      "state:(\"New York\" OR \"Georgia\" OR \"South Carolina\")":5,
      "state:(\"Illinois\" OR \"Texas\")":3,
      "state:(\"California\")":3},
    ...
  }
}
```

正如你所见，多个子查询通过查询分面可以组合成一个 Solr 请求。之前的例子是针对特定数据集的，因为查询的是所有已知的可能分面值。8.1 节其实已经给出了一个查询分面例子——基于价格的分面，这里的价格区间并不是均匀间隔的。我们使用测试数据重新创建这个例子，详见代码清单 8.8。

## 代码清单 8.8 多个价格区间的查询分面

## 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&rows=0&facet=true&
facet.query=price:[* TO 5]&
facet.query=price:[5 TO 10]&
facet.query=price:[10 TO 20]&
facet.query=price:[20 TO 50]&
facet.query=price:[50 TO *]
```

## 搜索结果

```
...
"response":{ "numFound":20, "start":0, "docs":[ ] },
"facet_counts":{
  "facet_queries":{
    "price:[* TO 5]":6,
    "price:[5 TO 10]":5,
    "price:[10 TO 20]":3,
    "price:[20 TO 50]":6,
    "price:[50 TO *]":0, ...
```

这个例子演示了查询分面在查询阶段如何针对任意 Solr 查询有效地创建新的信息区间。实际上，在 Solr 中可以很容易地创建一个新字段，例如，新建 `price_range` 字段存储每一段区间值。这一操作还需要基于新的 `price_range` 字段进行字段分面，找出 5 个区间段中各自包含的值。这也要求在 Solr 索引内容时使用这些区间分段规则。因此，如果区间段发生变化就必须重新索引文档，这个过程并不容易，特别是在 Solr 中积累了大量内容之后尤其不易。查询分面提供了一种很好的替代方法，在查询时可以完全自由地指定或重新定义哪些分面值需要计算和返回。

本节的例子虽然都很简单，但它们足以说明，基于任意查询的分面具有非常大的灵活性。Solr 提供了许多强大的查询功能，包括嵌套查询（参见第 7 章）和函数查询（参见第 15 章），基于分面的高级查询能够帮助你做到所能想到的一切。例如，利用 Solr 的半径查询，可以围绕特定地点创建同心圆（5 公里以内、5 ~ 10 公里、10 ~ 20 公里以及 20 公里以上），形成基于位置距离的分面。另外，将自定义相关度函数作为函数查询，可以基于函数计算值生成查询分面。分面的扩展能力非常强，根据特定搜索需求可以灵活设置分面。

虽然查询分面非常灵活，但它们也有可能对 Solr 请求造成负担。这是因为分面计数的生成所依赖的单一值都必须明确指定。下一节介绍 Solr 的区间分面功能，它让基于数值和日期的分面变得更加容易实现。

## 8.5 区间分面

顾名思义，区间分面将数值和日期字段值分成一些区间段，以便于 Solr 返回各

个区间以及它们包含的文档数。创建多个不同的查询分面来表示多个区间值的做法，可以用区间分面很好地替代。

上一节根据 8.2 节示例数据的价格字段创建了一个查询分面。搜索引擎返回的值均匀分布，类似的分面计数通过 Solr 的区间分面返回，详见代码清单 8.9。

代码清单 8.9 价格字段的区间分面示例

#### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&
facet.range=price&
facet.range.start=0&
facet.range.end=50&
facet.range.gap=5
```

#### 搜索结果

```
...
"response":{"numFound":20,"start":0,"docs":[ ]},
"facet_counts":{
  ...
  "facet_ranges":{
    "price":{
      "counts":[
        "0.0",6,
        "5.0",5,
        "10.0",0,
        "15.0",3,
        "20.0",2,
        "25.0",2,
        "30.0",1,
        "35.0",0,
        "40.0",0,
        "45.0",1],
      "gap":5.0,
      "start":0.0,
      "end":50.0}
    }}}
}}
```

这个例子的输出与代码清单 8.8 的结果相似，但有两点不同。首先，区间分面返回了查询落在 `facet.range.start` 和 `facet.range.end` 两个参数间的每个区间段的统计数，有些不包含文档的区间也被返回了。其次，与代码清单 8.7 的查询分面不同的是，根据 `facet.range.gap` 参数设置，区间分面的区间间隔是相同的。根据应用需求可以调整间隔，以创建更大或更小的区间段。返回所有区间段的做法会节省大量时间，因为不必设置许多 `facet.query` 参数就可以实现类似的效果。

区间分面还有其他一些参数可供选择，包括 `facet.range.hardend`、

`facet.range.other` 和 `facet.range.include`。表 8.2 列出了区间分面参数及其可用选项。

表 8.2 在 Solr 的 URL 中通过指定区间分面参数来修改分面行为

Solr 参数	可能取值	说明
<code>facet.range</code>	任何索引过的数值字段或时间字段的名称	确定对哪个字段计算区间分面。该参数可多次指定以返回多个分面
<code>facet.range.start</code>	区间左侧的数值或时间值	该参数指定区间的下限。低于下限的值不计入分面。该参数基于字段进行指定
<code>facet.range.end</code>	区间右侧的数值或时间值	该参数指定区间的上限，高于上限的值不计入分面。该参数可以基于字段进行指定
<code>facet.range.gap</code>	对于日期字段，DateMath 表达式 (+1DAY、+2MONTHS、+1HOUR 等)。对于数值字段，区间间隔为数字	区间的大小。为创建子区间，在下限 ( <code>facet.range.start</code> ) 到上限 ( <code>facet.range.end</code> ) 之间增加间隔。该参数基于字段进行指定
<code>facet.range.hardend</code>	true, false	如果间隔 ( <code>facet.range.gap</code> ) 在下限与上限之间不是均匀分布的，那么最后一个子区间的大小会与之前的子区间不同。若 <code>hardend</code> 为真，以上限结束，最后一个子区间可能较小。若 <code>hardend</code> 为假，最后一个子区间大小会超过上限，与其他子区间的大小相同，大小为间隔值。该参数基于字段进行指定
<code>facet.range.other</code>	before, after, between, all, none	指明区间中应包含的其他区间。before 选项为下限之前的所有值创建一个子区间。after 选项为上限之后的所有值创建一个子区间。between 选项为上限与下限之间的所有值创建一个子区间。该参数可以多次指定以包含多个值。all 选项包括 before、after 与 between 三个选项。如果 none 选项存在，则该参数会覆盖指定的任何其他参数。该参数基于字段进行指定
<code>facet.range.include</code>	lower, upper, edge, outer, all	lower 表示所有区间包含其下限。upper 表示所有区间包含其上限。edge 表示第一个子区间包含其下限，最后一个子区间包含其上限。outer 表示 before 与 after 子区间（来自 <code>facet.range.other</code> ）分别包含其下限与上限。该参数可以多次指定以包含多个值。all 可以分别指定参数中的每一个。该参数基于字段进行指定

从表 8.2 可以看出，Solr 的区间分面参数提供了丰富的选项。与字段分面一样，区间分面的一些参数使用 `f.<fieldName>.<FacetParameter>=<value>` 在字段上指定。

对数值和日期值进行分面时，相较于查询分面，区间分面提供了更方便和更简洁的查询语法。当区间分面变得过于复杂时，也可以反过来使用查询分面，以利用

Solr 的强大查询功能, 8.4 节曾讨论过 (更多内容参见第 15 章)。以上介绍了三种分面, 其中字段分面是使用最广泛的和最简单的分面类型。对于每一种分面类型, 我们已经了解了如何向 Solr 请求返回分面值。还一个问题没有讨论, 如果用户选择了一个分面, 如何进行后续的查询限定? 这是下一节要讨论的话题。

## 8.6 基于分面值的过滤

为了让用户根据分面对搜索结果进行探索, Solr 返回分面只是第一步。向用户展示分面值之后, 下一步是让用户点击一个或多个分面值, 将其作为过滤器使用。本节讨论这些过滤器的最佳使用方法。

### 8.6.1 在分面上使用过滤器

在初级水平时, 与向查询中添加额外的过滤器 (fq 参数) 相比, 在分面上使用过滤器没那么困难。假设在搜索中返回三个分面, 两个字段分面(州字段和城市字段) 和一个查询分面 (价格字段), 初始查询和搜索结果详见代码清单 8.10。

代码清单 8.10 基于 tags 字段的分面, 以餐馆数据为例

#### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&
facet.field=state&
facet.field=city&
facet.query=price:[* TO 10]&
facet.query=price:[10 TO 25]&
facet.query=price:[25 TO 50]&
facet.query=price:[50 TO *]
```

#### 搜索结果

```
...
"facet_counts":{
  "facet_queries":{
    "price:[* TO 10]":11,
    "price:[10 TO 25]":5,
    "price:[25 TO 50]":4,
    "price:[50 TO *]":0},
  "facet_fields":{
    "state":[
      "Georgia",6,
      "California",4,
      "New York",4,
      "Texas",3,
      "Illinois",2,
      "South Carolina",1],
```

所有的州都被包含进来了 (无过滤器)。



```

"city": [
  "Atlanta, GA", 6,
  "New York, NY", 4,
  "San Francisco, CA", 3,
  "Austin, TX", 3,
  "Chicago, IL", 2,
  "Greenville, SC", 1,
  "Los Angeles, CA", 1],
"facet_dates": {},
"facet_ranges": {}
...

```

所有的城市都被包含进来了（无过滤器）。

搜索结果也会返回，但没有显示在代码清单 8.10 中。搜索界面可能会向用户显示这个全面查询的可用分面值。在这个例子中，如何根据一个分面值进行过滤呢？其实你已经知道该怎么做了，在查询中为每个选中的分面值添加一个过滤器。代码清单 8.11 展示了第二轮搜索，用户点击了 California。

#### 代码清单 8.11 基于字段分面的过滤

##### 查询请求

```

http://localhost:8983/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field=state&
facet.field=city&
facet.query=price:[* TO 10]&
facet.query=price:[10 TO 25]&
facet.query=price:[25 TO 50]&
facet.query=price:[50 TO *]&
fq=state:California

```

影响搜索结果的过滤器。

##### 搜索结果

```

...
"facet_counts": {
  "facet_queries": {
    "price:[* TO 10]": 2,
    "price:[10 TO 25]": 0,
    "price:[25 TO 50]": 2,
    "price:[50 TO *]": 0,
  },
  "facet_fields": {
    "state": [
      "California", 4,
    ],
    "city": [
      "San Francisco, CA", 3,
      "Los Angeles, CA", 1,
    ],
    "facet_dates": {},
    "facet_ranges": {}
  }
}
...

```

最终的分面计数反映出搜索结果的数量减少。

代码清单 8.11 与代码清单 8.10 的不同之处在于，用户连续使用了分面。用户执行初始搜索得到一组分面，然后选择一个分面继续搜索，通过过滤器缩小搜索结果

范围。接着还可以继续执行第三轮搜索，详见代码清单 8.12。

代码清单 8.12 基于字段分面与查询分面的过滤

#### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*:*&facet=true&facet.mincount=1&
facet.field=state&
facet.field=city&
facet.query=price:[* TO 10]&
facet.query=price:[10 TO 25]&
facet.query=price:[25 TO 50]&
facet.query=price:[50 TO *]&
fq=state:California&
fq=price:[* TO 10]
```

在之前的分面上使用两个过滤器。

#### 搜索结果

```
...
"facet_counts":{
  "facet_queries":{
    "price:[* TO 10]":2,
    "price:[10 TO 25]":0,
    "price:[25 TO 50]":0,
    "price:[50 TO *]":0,
  },
  "facet_fields":{
    "state":[
      "California",2,],
    "city":[
      "San Francisco, CA",2,],
  },
  "facet_dates":{},
  "facet_ranges":{}
}
```

使用的过滤器会影响分面计数。

这里使用了 `state:California` 和 `price:[* TO 10]` 两个过滤器，搜索结果得到了进一步细化。与查询限定相匹配的搜索结果只有两家加州餐馆，而且两家餐馆都位于加州的旧金山市。

值得注意的是，这里每个例子处理的都是单一值字段的分面。因此，在该分面值上进行过滤，就不会返回与其他分面值匹配的文档。这对单值字段来说是有意义的。如果一个文档只能拥有一个价格或只能出现在一个州，那么它就不会出现在选择了其他价格分面和州分面的情况中。然而，并不是所有的字段都只包含单一值。在 Solr 的 `schema.xml` 文件中，字段被标记为多值或要被分词，这两种情况都会对分面产生多个词项。以餐馆索引的多值标签字段为例，我们来看看如何实现多值字段的分面。代码清单 8.13 是对标签字段连续使用过滤器进行的多次搜索。

## 代码清单 8.13 对包含多值的分面字段使用多个过滤器

## 不使用过滤器

## 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
  facet.field=name&
  facet.field=tags
```

## 搜索结果

```
...
"facet_fields":{
  "name":[
    "Starbucks",6,
    "McDonalds",5,
    "Pizza Hut",3,
    "Red Lobster",3,
    "Freddy's Pizza Shop",1,
    "Sprig",1,
    "The Iberian Pig",1],
  "tags":[
    "breakfast",11,
    "coffee",11,
    "sit-down",8,
    "fast food",5,
    "hamburgers",5,
    "wi-fi",5,
    "pizza",4,
    "delivery",3,
    "sea food",3,
    "gluten-free",1,
    "pasta",1,
    "sit down",1,
    "southern cuisine",1,
    "spanish",1,
    "tapas",1,
    "upscale",1]},
...

```

11 家餐馆提供咖啡, 5 家餐馆提供汉堡包, 5 家餐馆提供 wi-fi。

## 使用一个过滤器

## 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
  facet.field=name&facet.field=tags&fq=tags:coffee
```

## 搜索结果

```
...
"facet_fields":{
  "name":[
    "Starbucks",6,
    "McDonalds",5],
  "tags":[
    "breakfast",11,

```

```
"coffee", 11,  
"fast food", 5,  
"hamburgers", 5,  
"wi-fi", 5}],  
...
```

使用 coffee 过滤器之后, 5 家提供汉堡包和 wi-fi 的餐馆被保留了下来。

### 使用两个过滤器

#### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&facet.mincount=1&  
facet.field=name&facet.field=tags&fq=tags:coffee&fq=tags:hamburgers
```

#### 搜索结果

```
...  
"facet_fields": {  
  "name": [  
    "McDonalds", 5],  
  "tags": [  
    "breakfast", 5,  
    "coffee", 5,  
    "fast food", 5,  
    "hamburgers", 5,  
    "wi-fi", 5],  
  ...  
}
```

所有提供汉堡包的餐馆也提供咖啡与 wi-fi。

从代码清单 8.13 可以看出, 多值字段可以继续返回其他值, 只要包含这些值的文档仍然能匹配所有在用的过滤器即可。值得注意的是, 这些例子对每个选定的过滤器值在 Solr 的 URL 中指定了各自的 `fq` 参数, 但其实没必要这样做。事实上, 代码清单 8.13 中最后一个搜索的过滤器 `fq=tags:coffee&fq=tags:hamburgers` 可以轻松地转换为 `fq=tags:(coffee AND hamburgers)`, 或者其他等价的布尔逻辑表达式。这种转换需要在 Solr 的过滤器缓存中少数几次查找 (参见第 4 章), 而且还提供了与过滤器值交互的更多控制。

不要使用“并”(AND) 逻辑来连接每个分面过滤器, 使用“或”(OR) 逻辑来连接分面过滤器才是正确有效的使用方法。例如, 当用户选择多个城市时, 过滤出选中的那些城市。8.7 节会讨论多选分面, 教你如何在分面中一次选中多个过滤器, 这种方法对单值字段也适用。

## 8.6.2 基于分面值的安全过滤方法

以上的分面过滤器示例都是基于单值词项的, 例如, `coffee` 或 `hamburgers`。事实上, 分面返回的词项在处理上更有难度, 例如, 短语型词项的处理。举例来说, 如何对短语型词项 `Los Angeles` 进行分面? 你可能已经知道, 过滤器 `fq=city:Los Angeles` 是无效的。因为从语法上看, 这个过滤器需要在默认的待查字段中寻找包含 `Los` 城市和 `Angeles` 词项的文档。为了处理以空格分隔的短语, 大多数 Solr 开发者会使用引号把分面的所有词项括起来。因此, 查询语法应该

是这样的 `fq=city:"Los Angeles"`。

不过,盲目地使用引号把要过滤的项括起来可能也会遇到问题。如果项本身包含引号,那么不做转义处理的话,语法就会出错。因此,如果搜索歌曲集索引,对 `the "in" crowd` 这首歌进行分面,就需要使用转义符才能将项更安全地传递给 Solr,正确的过滤器语法为 `fq=name:"the \"in\" crowd"`。除了注意引号和转义引号之外,还必须留意在分面的字段上进行的所有文本处理操作。正如第 6 章曾介绍过,基于字段的文本分析在内容索引与查询两个阶段可以分别定义。因此,如果配置存在匹配失误(通常不是好的做法),这可能说明,尝试过滤的值与分面返回的文档数不一致。

所幸,我们有一个非常简单的解决方法,它能够确保分面返回值与随后的过滤器值两者找到准确一致的文档,即使用词项查询解析器(`TermQParserPlugin`),参见第 7 章。此查询解析器的一个好处在于,绕过了字段上已定义的文本分析链,直接将项与 Solr 索引进行匹配。这样做节省了文本处理时间,并且避免了之前讨论过的引号和转义逻辑等问题。词项查询解析器在 `Los Angeles` 上的使用语法为 `fq={!term}Los Angeles`,在歌曲 `the "in" crowd` 上的使用语法为 `fq={!term}the "in" crowd`。

对所有分面过滤器使用词项查询解析器的做法存在一个缺点,即它不支持布尔语法。因此,如果要在一个过滤器里组合多个分面值,那么需要使用嵌套查询语法(参见第 7 章)。代码清单 8.14 演示了这两种使用方法:一种是分别为每个分面项使用过滤器;另一种是通过词项查询解析器把这些分面项组合成为一个过滤器。

#### 代码清单 8.14 使用词项查询解析器对分面值进行过滤

##### 方法 1: 每个词项对应一个过滤器

###### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field=name&facet.field=tags&
fq={!term f=tags}coffee&fq={!term f=tags}hamburgers
```

##### 方法 2: 所有词项对应一个过滤器

###### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&
facet=true&facet.mincount=1&facet.field=name&facet.field=tags&
fq=_query_:"{!term f=tags}coffee" AND _query_:"{!term f=tags}hamburgers"
```

无论使用哪一种方式实现分面过滤器,查询解析过滤器都会让查询速度变得更快一些。如果选择使用第二种方法——嵌套查询语法,由于整个嵌套查询包含在引号中,所以需要在项中添加转义引号。如果在分面过滤器中使用布尔逻辑,那么这种方式可能会造成一些麻烦。

本节中我们已经看到，在分面上使用过滤器与过滤器的其他使用没有什么差异。在每个分面上分别使用过滤器或在多个分面上使用一个过滤器，这两种方法都是可行的。此外，在使用分面过滤器时，通过词项查询解析器绕过文本处理，可以避免难以处理的字符转义问题。至此，我们学会了各种基础类型分面的请求与过滤操作，还有更多内容有待深入学习。下一节介绍分面使用的一些其他方法，比如，出于显示目的的分面重命名，以及对过滤掉的文档进行分面计数等。

## 8.7 多选分面、键与标记

向 Solr 请求分面时，显示给用户看的分面名称并不总是那么直观易懂，即使在搜索应用技术系统中处理分面名称也可能存在这样的问题。Solr 提供了方便的分面重命名方法，可以在分面返回时对其进行重命名，使之更符合应用场景的理解和使用。Solr 还可以对过滤掉的文档进行分面计数，这对多选分面而言非常有用。对搜索结果进行过滤后，仍然可以看到被过滤掉的、在过滤器应用之前与查询匹配的相关文档数。本节介绍键 (key)、标记 (tag) 与排斥 (exclude) 局部参数 (第 7 章介绍过局部参数)，利用它们来实现分面重命名与分面多选功能。

### 8.7.1 键

所有的分面都有名称，方便开发者进行识别。默认情况下，字段分面与区间分面的分面名称就是字段名，查询分面的分面名称是基于分面值 and 计数的查询本身。使用键这个局部参数可以方便地对分面进行重命名，详见代码清单 8.15 所示。

代码清单 8.15 分面重命名

#### 默认的 Solr 分面名称

##### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field=city&
facet.query=price:[* TO 10]&
facet.query=price:[10 TO 25]&
facet.query=price:[25 TO 50]&
facet.query=price:[50 TO *]
```

字段分面命名默认  
使用字段名称。

##### 搜索结果

```
...
"facet_counts":{
  "facet_queries":{
    "price:[* TO 10]":11,
    "price:[10 TO 25]":5,
    "price:[25 TO 50]":4,
    "price:[50 TO *]":0,
  }
```

查询分面命名默认  
使用查询本身。

```
"facet_fields":{
  "city":[
    "Atlanta, GA",6,
    "New York, NY",4,
    "Austin, TX",3,
    "San Francisco, CA",3,
    "Chicago, IL",2,
    "Greenville, SC",1,
    "Los Angeles, CA",1]},
  "facet_dates":{},
  "facet_ranges":{}}
```

← 查询分面命名默认  
使用查询本身。

### 通过明确指定 key 值对分面进行重命名

#### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field={!key="Location"}city&
facet.query={!key="<$10"}price:[* TO 10]&
facet.query={!key="$10 - $25"}price:[10 TO 25]&
facet.query={!key="$25 - $50"}price:[25 TO 50]&
facet.query={!key=">$50"}price:[50 TO *]
```

#### 搜索结果

```
...
"facet_counts":{
  "facet_queries":{
    "<$10":11,
    "$10 - $25":5,
    "$25 - $50":4,
    ">$50":0},
  "facet_fields":{
    "Location":[
      "Atlanta, GA",6,
      "New York, NY",4,
      "Austin, TX",3,
      "San Francisco, CA",3,
      "Chicago, IL",2,
      "Greenville, SC",1,
      "Los Angeles, CA",1]},
    "facet_dates":{},
    "facet_ranges":{}}
```

为查询分面赋  
予更容易理解  
的名称。

← 字段分面名  
称由 city 改  
为 Location。

分面重命名适合于很多应用场景。它允许搜索应用请求查询分面，例如，在后置处理阶段不需要搜索应用对结果集的查询进行解析。它还能对分面赋予用户易于理解的名称，不管分面的底层字段与查询怎么样，让搜索界面上的结果显示更加直观。此外，每个分面名称是唯一的，通过键的指定，在相同字段上可以定义不止一个分面（这对字段分面和区间分面而言是有用的），返回的每个分面都有唯一的名称。这种方法的优点是，根据查询阶段的规则，它可以让多个字段映射成同一个名称。



假设，搜索索引有两个字段：SecretInformationOnlyAvailableToSomeUsers 和 InformationAvailableToAllUsers。根据这样的设置，在查询阶段创建一个分面，指定 facet.field={!key="Information"}InformationAvailableToAllUsers 或 facet.field={!key="Information"}SecretInformationOnlyAvailableToSomeUsers。这种间接层在许多应用场景中使用起来非常方便。例如，想在任何时候重新定义分面，让它指向一个不同的字段，使用这种间接层对应用技术体系的改动将会很小。除了分面重命名，Solr 还提供了特定过滤器的标记功能，便于控制过滤器与其他 Solr 功能的交互。

8.7.2 标记、排除和多选分面

在 Solr 查询请求中使用过滤器，搜索结果必然会包含每个过滤器。默认情况下，分面也要这样处理。但是，这里有一个问题，我们经常需要查看已经被查询排除在外的搜索结果的分面计数。图 8.8 讨论了这个问题，在 8.2 节的餐馆示例数据上请求一些分面，并使用“加州”这个过滤器。

分面过滤器应用之前

<p>▶ 州</p> <p><input type="checkbox"/> Georgia (6)</p> <p><input type="checkbox"/> California (4)</p> <p><input type="checkbox"/> New York (4)</p> <p><input type="checkbox"/> Texas (3)</p> <p><input type="checkbox"/> Illinois (2)</p> <p>...</p>	<p>▶ 价格区间</p> <p><input type="checkbox"/> &lt; \$5 (6)</p> <p><input type="checkbox"/> \$5 - \$10 (5)</p> <p><input type="checkbox"/> \$10 - \$20 (3)</p> <p><input type="checkbox"/> \$20 - \$50 (6)</p> <p><input type="checkbox"/> \$50+ (0)</p>	<p>▶ 城市</p> <p><input type="checkbox"/> Atlanta, GA (6)</p> <p><input type="checkbox"/> New York, NY (4)</p> <p><input type="checkbox"/> Austin, TX (3)</p> <p><input type="checkbox"/> San Francisco, CA (3)</p> <p><input type="checkbox"/> Chicago, IL (2)</p> <p>...</p>
--	---	--

“州:加州”分面过滤器应用之后

<p>▶ 州</p> <p><input checked="" type="checkbox"/> California (4)</p>	<p>▶ 价格区间</p> <p><input type="checkbox"/> &lt; \$5 (0)</p> <p><input type="checkbox"/> \$5 - \$10 (2)</p> <p><input type="checkbox"/> \$10 - \$20 (0)</p> <p><input type="checkbox"/> \$20 - \$50 (2)</p> <p><input type="checkbox"/> \$50+ (0)</p>	<p>▶ 城市</p> <p><input type="checkbox"/> San Francisco, CA (3)</p> <p><input type="checkbox"/> Los Angeles, CA (1)</p>
--	---	---

图 8.8 默认情况下，选择一个分面进行过滤后，分面值不再包含已经过滤掉的文档。如果希望让用户在搜索中选择多个分面值，由于分面过滤掉的值不再出现在选项中了，所以无法使用逻辑“或”(OR)来得到这些值。这个问题值得思考

虽然期望的搜索结果在之前的搜索界面上仅显示加州地区的餐馆，但是如果在分面中继续显示其他州的话，就可以进一步扩展查询。同样的道理也适用于价格区间分面与城市分面。让用户每次在一个分面上只搜索一个值，这样做是不明智的。

所幸，Solr 提供分面排除功能来解决这一问题。分面排除可以添加被搜索请求中任何过滤器过滤掉的文档。将过滤掉的文档数计入分面数，这样就可以对每个分

面进行有效计数，同时忽略该分面上已使用的过滤器。tag 与 ex 是实现这一方法的两个必要局部参数。代码清单 8.16 演示了如何使用这两个参数对图 8.8 中的例子进行多选分面。

代码清单 8.16 使用 tag 与 ex 实现多选分面

#### 查询请求

```
http://localhost:8983/solr/restaurants/select?q=*&facet=true&facet.mincount=1&
facet.field={!ex=tagForState}state&
facet.field={!ex=tagForCity}city&
facet.query={!ex=tagForPrice}price:[* TO 5]&
facet.query={!ex=tagForPrice}price:[5 TO 10]&
facet.query={!ex=tagForPrice}price:[10 TO 20]&
facet.query={!ex=tagForPrice}price:[20 TO 50]&
facet.query={!ex=tagForPrice}price:[50 TO *]&
fq={!tag="tagForState"}state:California
```

state 分面应忽略标记为 tagForState 的过滤器。

查询结果限制在加州。

#### 搜索结果

```
...
"facet_counts":{
  "facet_queries":{
    "{!ex=tagForPrice}price:[* TO 5]":0,
    "{!ex=tagForPrice}price:[5 TO 10]":2,
    "{!ex=tagForPrice}price:[10 TO 20]":0,
    "{!ex=tagForPrice}price:[20 TO 50]":2,
    "{!ex=tagForPrice}price:[50 TO *]":0,
  "facet_fields":{
    "state":[
      "Georgia",6,
      "California",4,
      "New York",4,
      "Texas",3,
      "Illinois",2,
      "South Carolina",1],
    "city":[
      "San Francisco, CA",3,
      "Los Angeles, CA",1]],
    "facet_dates":{},
    "facet_ranges":{}}
  ...
```

state 分面忽略了 state:California 上的过滤器。

其他分面不会忽略 state:California 上的过滤器。

正如你所见，过滤器将查询限制在加州范围，但是与查询其余部分相匹配的文档数依然按照每个州返回，而不仅仅是显示加州的文档数。图 8.9 显示了多选分面结果与初始未过滤分面结果的对比。

分面过滤器应用之前		
州 <input type="checkbox"/> Georgia (6) <input type="checkbox"/> California (4) <input type="checkbox"/> New York (4) <input type="checkbox"/> Texas (3) <input type="checkbox"/> Illinois (2) ...	价格区间 <input type="checkbox"/> < \$5 (6) <input type="checkbox"/> \$5 - \$10 (5) <input type="checkbox"/> \$10 - \$20 (3) <input type="checkbox"/> \$20 - \$50 (6) <input type="checkbox"/> \$50+ (0)	城市 <input type="checkbox"/> Atlanta, GA (6) <input type="checkbox"/> New York, NY (4) <input type="checkbox"/> Austin, TX (3) <input type="checkbox"/> San Francisco, CA (3) <input type="checkbox"/> Chicago, IL (2) ...
“州:加州”分面过滤器应用之后		
州 <input type="checkbox"/> Georgia (6) <input checked="" type="checkbox"/> California (4) <input type="checkbox"/> New York (4) <input type="checkbox"/> Texas (3) <input type="checkbox"/> Illinois (2) ...	价格区间 <input type="checkbox"/> < \$5 (0) <input type="checkbox"/> \$5 - \$10 (2) <input type="checkbox"/> \$10 - \$20 (0) <input type="checkbox"/> \$20 - \$50 (2) <input type="checkbox"/> \$50+ (0)	城市 <input type="checkbox"/> San Francisco, CA (3) <input type="checkbox"/> Los Angeles, CA (1)

图 8.9 通过 state:California 进行分面过滤，除了州分面之外其他字段的分面值都根据过滤条件进行变化。由于州分面被标记为 tagForState，所以它的分面值没有发生变化。这样做可以让用户继续选择其他州，达到多选分面的搜索效果

就查询机制而言，tagForState 作用于加州的过滤器，实现的语法为 `fq={!tag="tagForState"}state:California`。当请求州分面时，排除所有标记为 tagForState 的过滤器，实现的语法为 `facet.field= {!ex=tagForState}state`。

你可能无法看到代码清单 8.16 的搜索结果，但有一点要搞清楚：由于查询中使用了这个过滤器，所以即便州分面没有受到该过滤器的限制，但 Solr 返回的文档仍然会限制在加州。出于同样的原因，没有标记为 tagForState 的其他所有分面也受限于加州这个过滤器。

另外还需注意，其他分面（如城市和价格）也可能包含排除标记，但它们没有与任何标记过的过滤器相对应。这些排除标记不是必需的，它们不会造成什么问题，也可以被忽略。如果要重新执行这个查询，并在另一个分面值上增加一个过滤器（使用一个排除标记），那么当前未使用的排除标记就会发挥作用。在任何包含排除标记的过滤器出现之前，是否对分面使用排除标记，由你自己决定。如果使用排除标记，语法上可能会显得冗余，但不会对返回的搜索结果造成负面影响。

将标记与分面进行混搭，有可能实现有趣的搜索界面和提供一些数据分析功能，不过这些内容已经超出了本章范围。如果想要了解更多，请自由尝试 Solr 的这些功能。

至此，你已经学习了 Solr 最常见的分面功能，包括字段分面、查询分面与区间分面。下一节会简要介绍分面的一些高级功能，更多细节内容会在第 15 章讨论复杂查询操作时展开。

## 8.8 超越分面基础

本章简要介绍了 Solr 最常用的分面功能，这只是拉开了分面的序幕。分面会大量使用 Solr 的缓存，因此需要对 Solr 内置缓存的使用方式进行优化，以期最大限度发挥分面请求的性能。第 12 章会进一步探讨分面的缓存问题。

除了性能优化，第 15 章会介绍更多的高级分面形式。有一种高级分面形式称为透视分面，它提供了多维度的分面功能。如果只知道标签字段排名靠前的几个值是不够的，还需要知道城市排名靠前的标签，这样的情况该如何处理？一种方式是先进行一轮搜索，得到城市字段所有值的分面，接下来搜索每个城市的标签分面。不过，这种方式不能很好地扩展，很容易导致执行数十个或上百个搜索，文档集也会变得很庞大。Solr 提供透视分面来解决这个问题。透视分面可以在多个维度上进行分面，使用一个搜索返回各个维度上的计算值。要了解高级分面的更多内容，请参见第 15 章的 15.3 节。

## 8.9 本章小结

恭喜你读完了颇有深度的一章，这一章介绍了 Solr 最强大的功能之一。正如你所见，分面提供了一种快捷方式，让用户能够快速掌握与查询匹配的文档的大致情况。现如今，搜索驱动的主流网站几乎都提供分面，帮助用户进行探索式搜寻。Solr 的字段分面可以返回每个字段靠前的值，区间分面返回数字与时间值的区间段，查询分面返回任意复杂查询的文档数。

你已经知道如何在分面返回时使用键对其重命名，使用标记与分面排除来实现多选分面和返回查询过滤掉文档数。除此之外，本章还介绍了当用户点击分面时，对应地在查询上设置过滤器的各种方法。

最后，本章提到了 Solr 的多维透视分面功能，详细内容会在第 15 章的复杂查询操作中进一步讨论。读到此处，你应该能够使用 Solr 的各种分面方法实现一些复杂的搜索应用了。

下一章介绍 Solr 的另一个常见功能——搜索结果高亮。在搜索结果返回时，将匹配到的文本片段显示在搜索结果列表中，向用户提供潜在的洞察力，帮助用户判断搜索结果中的文档是否值得进一步去探索。

# 搜索结果高亮

## 本章要点

- 在搜索结果中高亮显示查询词项
- 为每个搜索结果选择最佳片段进行显示
- 使用可选参数改善高亮效果
- 使用 `FastVectorHighlighter` 提升高亮性能
- 使用 Solr 最新的高亮实现方法 `PostingsHighlighter`

本章介绍 Solr 的一个核心功能——搜索结果高亮（以下简称“高亮”），它能实现查询词项在搜索结果中高亮显示的效果。高亮功能帮助用户快速浏览结果，确定哪些结果值得进一步研究，是否应该导航到下一页，亦或是执行不同查询。

为了使高亮的学习更有趣，本章使用由 Infochimps ([www.infochimps.com/datasets/60000-documented-ufo-sightings-with-text-descriptions-and-metada](http://www.infochimps.com/datasets/60000-documented-ufo-sightings-with-text-descriptions-and-metada)) 提供的一个免费数据集，该数据集包含数千条美国各地不明飞行物（UFO）的目击数据。本章的 UFO 搜索应用允许用户输入少量有关目击情况的关键词，查询是否曾有类似的目击情况出现。

本章之所以选择这个例子，因为它提供了本书尚未谈及的两个学习要点。首先，与前面章节中的数据直接导入示例所不同的是，如果不做一些前期的数据建模和预处理，就不能将 UFO 数据集加载到 Solr 中。构建搜索应用时，一个基本的要求就

是将原始数据转换成能够被索引的数据格式。其次,这个例子可供读者以本书已讲的知识内容为基础来构建搜索应用。具体来说,要为 UFO 数据集构建索引(第 5 章),进行文本分析(第 6 章),在 `solrconfig.xml` 文件中配置搜索组件(第 4 章),探索有关目击数据的分面形式(第 8 章),并查询相似的目击结果(第 7 章)。本章的重点是高亮,但也要借助前面章节介绍的工具来构建搜索应用。

虽然说起来很简单,但 Solr 的高亮功能中有很多可选的配置参数, Solr 新手不是很清楚何时要使用哪个参数,有时会感到困惑。此外, Solr 提供了三种不同的开箱即用的高亮实现方法,这让人很难决定到底需要哪种方法。本章介绍这三种高亮实现方法,并提供何时使用哪种方法的参考建议。首先,我们来看如何在 UFO 搜索应用中使用高亮。

## 9.1 高亮简介

试想下雨天在天空中看到了一个蓝色火球,我们使用 UFO 搜索应用去查找类似的目击情况。高亮有助于判断搜索结果中的目击情况是否与查询相关。如图 9.1 所示,用户查询 `blue fireball in the rain` 来查找类似的 UFO 目击情况。

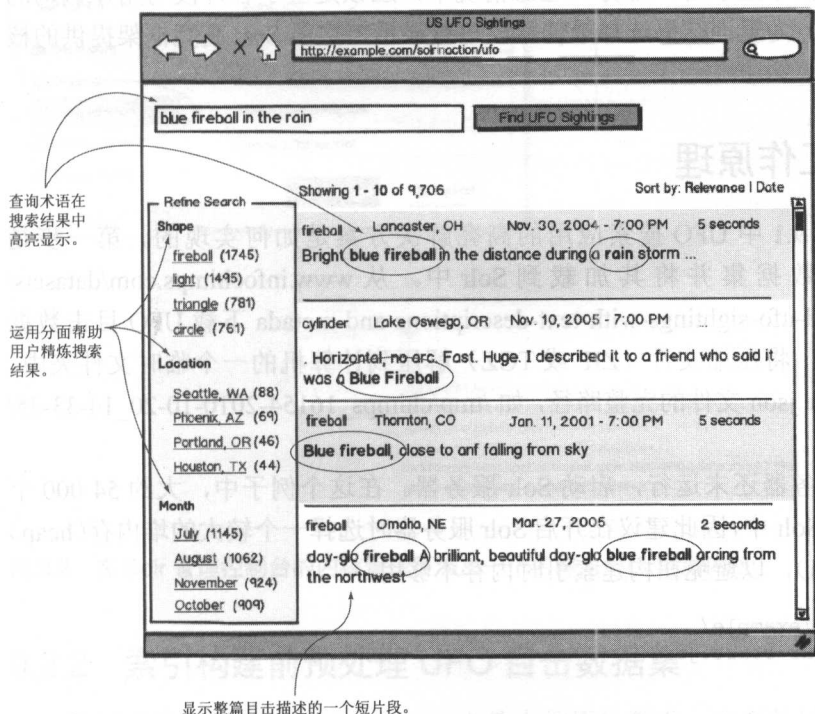


图 9.1 虚构的 UFO 搜索应用高亮显示结果截图

在图 9.1 中，UFO 搜索应用将查询词项在搜索结果中高亮显示。用户搜索 `blue fireball in the rain` 时，与查询词项 `blue`、`fireball` 和 `rain` 相匹配的搜索结果被加粗并高亮显示。基于这个设计好的示例，你可能会觉得，高亮显示搜索结果文档中的 `blue`、`fireball` 和 `rain` 很容易做到，使用简单的模式匹配和 HTML 标记就可以实现。但是远没有这么简单，Solr 还选择了最能代表用户查询的一小部分文档，即搜索结果片段。在图 9.1 中，每个结果所显示的文本都是基于用户的查询动态选择，而不是目击情况的完整描述。例如，在图 9.1 中的第一个目击情况的完整描述如下：

```
Bright blue fireball in the distance during a rain storm. Not a lightning storm, no thunder. No relevant towers or objects in the area.
```

需要明确的是，高亮显示查询词项只是 Solr 高亮功能中的很小一部分。这个功能的强大之处在于，它决定了每个搜索结果的哪些文本应当被高亮显示。

大多数的搜索应用都与图 9.1 中的情况类似，搜索结果显示的屏幕空间有限。如果文档很短并可以在结果列表中显示全部内容，对屏幕空间显示就不会构成太大的问题。但大多数情况下都只能显示每个结果文档的一小部分。这就提出一个问题：如何决定结果文档中显示哪一部分？理想情况下，应该是基于各片段与用户查询的匹配程度来决定。为查询结果选择最佳片段进行显示，这是 Solr 高亮框架提供的核心功能。

## 9.2 高亮工作原理

我们来看图 9.1 中 UFO 搜索应用的高亮解决方案是如何实现的。第一步是获得 UFO 目击数据集并将其加载到 Solr 中。从 [www.infochimps.com/datasets/60000-documented-ufo-sightings-with-text-descriptions-and-metada](http://www.infochimps.com/datasets/60000-documented-ufo-sightings-with-text-descriptions-and-metada) 下载 UFO 目击数据集。下载完成后，将压缩文件（ZIP 或 TGZ）解压到计算机的一个临时文件夹中。记录 `ufo_awesome.json` 文件的完整路径，如 `/tmp/chimps_16154-2010-10-20_14-33-35/ufo_awesome.json`。

如果 Solr 服务器还未运行，启动 Solr 服务器。在这个例子中，大约 54 000 个文档会被添加到 Solr 中，因此建议在开启 Solr 服务器时选择一个较大的堆内存(heap)大小（`-Xmx512m`），以避免在构建索引时内存不够用。

```
cd $SOLR_INSTALL/example/  
java -Xmx512m -jar start.jar
```



### 9.2.1 为 UFO 目击数据创建新的 Solr 内核

现在使用第 4 章中学到的知识创建一个新内核，代替 collection1 内核。在本章中将新内核命名为 ufo。要完成这些操作，需要复制 `$SOLR_INSTALL/example/` 目录下的 `collection1/` 文件夹，得到一个新的命名为 `ufo/` 的文件夹。现在不需要更改其他任何配置，但建议删除 `ufo/` 下的 `data` 文件夹，保证开始运行时新内核的索引为空。

```
cd $SOLR_INSTALL/example/solr
cp -r collection1 ufo
rm -rf ufo/data
rm ufo/core.properties
```

在 Solr 管理控制台的内核管理页单击 Add Core 按钮。如图 9.2 所示，填写表单。

在内核管理页面点击 Add Core 按钮，创建 ufo 内核。

从拷贝自集合 1 的 ufo 目录中添加新的 ufo 内核。

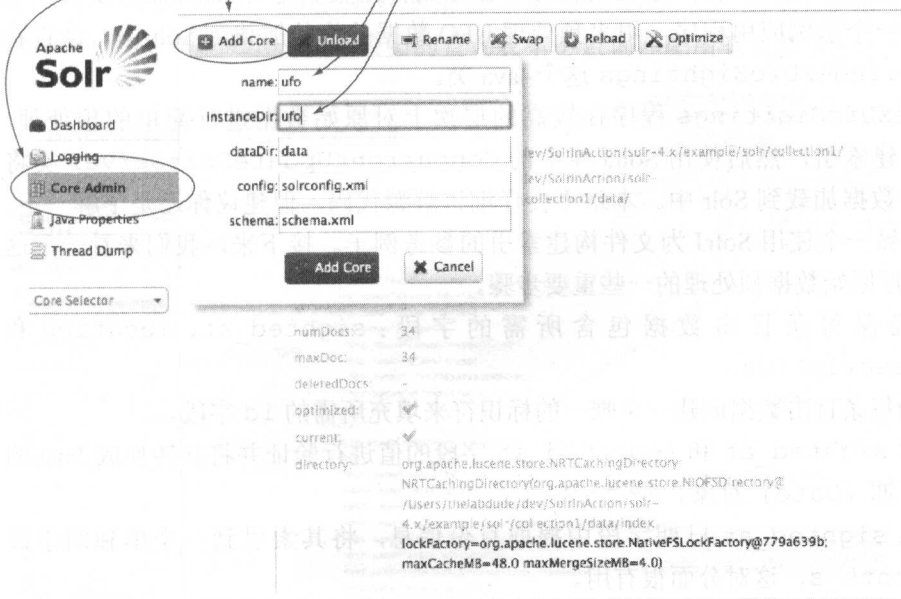


图 9.2 在 Solr 管理控制台的内核管理页添加一个名称为 ufo 的新内核

### 9.2.2 索引构建前预处理 UFO 目击数据集

我们来看一下 `ufo_awesome.json` 文件中的原始数据，本章使用这些数据来构建

索引。该文件的前两行内容如下：

```
{"sighted_at": "19951009", "reported_at": "19951009", "location": " Iowa City, IA", "shape": "", "duration": "", "description": "Man repts. witnessing &quot;flash, followed by a classic UFO, w/ a tailfin at back.&quot; Red color on top half of tailfin. Became triangular."}

{"sighted_at": "19951010", "reported_at": "19951011", "location": " Milwaukee, WI", "shape": "", "duration": "2 min.", "description": "Man on Hwy 43 SW of Milwaukee sees large, bright blue light streak by his car, descend, turn, cross road ahead, strobe. Bizarre!"}
```

每行是一个 JSON 对象，包含了一个 UFO 目击情况的所有描述字段。书中多次提到过，Solr 支持原生的 JSON 格式。但是，这些数据不符合由 collection1 内核复制而来的 ufo 内核中 schema.xml 文件的要求。因此，不能直接将这个 JSON 文件加载到 Solr，自动进行创建索引。思考一下，UFO 目击数据集哪里不符合 ufo 内核中 schema.xml 文件的要求。提醒一下，可以从 Solr 管理控制台查看 schema.xml 文件。

本例需要编写一个客户端应用程序，让原始目击数据符合 schema 的要求。书中提供了一个示例应用程序，用来预处理 UFO 数据并将其加载到 Solr 中。请查看 `sia.ch9.IndexUfoSightings` 这个 Java 类。

`IndexUfoSightings` 程序在较高的层次上对原始数据进行简单的预处理，以准备构建索引，然后使用 SolrJ 库中的 `ConcurrentUpdateSolrServer` 类将 UFO 目击数据加载到 Solr 中。本章不会仔细讲解源代码，但建议你还是了解一下，因为它是另一个使用 SolrJ 为文件构建索引的参考例子。接下来，我们来看一下这个程序进行原始数据预处理的一些重要步骤。

1. 确保每条目击数据包含所需的字段：`sighted_at`、`location` 和 `description`。
2. 为每条目击数据创建一个唯一的标识符来填充所需的 `id` 字段。
3. 对 `sighted_at` 和 `reported_at` 字段的值进行验证并将其转换成 Java 的日期 (`Date`) 对象。
4. 从 `sighted_at` 日期字段中提取月份信息，将其索引到一个单独的字段 `month_s`，这对分面很有用。
5. 将 XML 转义字符替换为它所表示的字符，例如，将 `&quot;` 替换为双引号 (`"`) 字符。
6. 将目击描述为空的字段变成一个空格字符。
7. 将 `location` 字段分为 `city` 和 `state` 两个字段。
8. 将 JSON 文件中存放目击信息的字段名转换成符合 Solr 动态字段要求的形式。例如，`shape` 变为 `shape_s`，`description` 变为 `sighting_en`。

9. 添加字词和句子末尾之间丢失的空格。例如, fog.We 变为 fog. We。这会改善目击描述数据的显示效果。

如果数据集来源于用户生成的内容,有时需要在进行索引的客户端代码中做一些数据预处理,以提升搜索应用的用户体验。

对客户端应用程序所进行的处理有了基本了解之后,我们使用它来构建 Solr 中的索引。将目录更改到 \$SOLR\_IN\_ACTION/ 目录下,执行如下操作:

```
java -jar solr-in-action.jar ufo -jsonInput
➡ $FULL_PATH_TO_UFO_SIGHTINGS_DIR/ufo_awesome.json
```

这个构建索引的应用程序运行速度非常快,不到一分钟就能运行完成。下面是程序运行结束时的输出,它表明有 54 190 条目击数据被加载到 Solr 中并进行了索引。

```
INFO [main] (IndexUfoSightings.java:106) - Sent 54190 sightings (skipped
6877) took #.# seconds
```

请注意,由于验证错误,忽略了 6877 条数据。现在对 ufo 内核执行一个匹配所有文档的查询,查看有多少条目击数据被索引了。图 9.3 显示了在 <http://localhost:8983/solr/#/ufo/query> 页面的查询表单中查询匹配所有文档(\*:\*)的结果。

ufo 内核的查询表单

匹配到所有文档的查询

仅找到 54 170 个文档!  
UFO 数据集中存在 20 个  
重复目击记录。

Request-Handler (q)

/select

common

q

fq

sort

start, rows

0 10

fi

df

Raw Query Parameters

key1=vsl3&key2=vsl2

wt

xml

indent

debugQuery

HTTP http://localhost:8983/solr/ufo/select?q=\*:\*&wt=xml&indent=true

```
<?xml version="1.0" encoding="UTF-8">
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="indent">true</str>
      <str name="q">*:*</str>
      <str name="_">1376589623880</str>
      <str name="wt">xml</str>
    </lst>
  </responseHeader>
  <result name="response" numFound="54170" start="0">
    <doc>
      <str name="id">19951009/19951009/IowaCity/Ia/7/8571878d36531869dc4c17dcd8b0113</str>
      <date name="sighted_at_dt">1995-10-09T06:00:00Z</date>
      <str name="month_s">October</str>
      <date name="reported_at_dt">1995-10-09T06:00:00Z</date>
      <str name="city_s">Iowa City</str>
      <str name="state_s">IA</str>
      <str name="location_s">Iowa City, IA</str>
      <str name="sighting_en">
        <str>Man repes. witnessing "flash, followed by a classic UFO, w/ a tailfin at back."
      </str>
      <long name="version">1443458798636236000</long>
    </doc>
    <str name="id">19951010/19951011/Milwaukee/Wi/7/bab1a1f44cdf0208d4efaf17447b76c8</str>
    <date name="sighted_at_dt">1995-10-10T06:00:00Z</date>
  </result>
</response>
```

图 9.3 对 ufo 内核查询匹配所有文档 (\*:\*) 的结果显示 54 170 个文档被索引; 客户端应用程序加载了 54 190 个文档, 其中存在 20 个重复文档

9.2.3 探索 UFO 目击数据集

表 9.1 介绍了由 IndexUfoSightings 应用程序处理生成的 UFO 目击数据文件情况。

表 9.1 UFO 目击数据文档

字段名	字段类型	说明	举例
Id	字符串	从其他字段生成的合成唯一标识符	20041130/20041204/ ...
sighted_at_dt	日期	UFO 目击日期 / 时间	2004-11-30T07:00:00Z
reported_at_dt	日期	向当局报告目击的日期 / 时间	2004-12-04T07:00:00Z
month_s	字符串	目击发生的月份	11 月
city_s	字符串	目击发生的美国城市	兰卡斯特 Lancaster
state_s	字符串	目击发生的美国州	俄亥俄州 OH
location_s	字符串	目击发生的美国州与城市	Lancaster, OH
shape_s	字符串	UFO 的形状	火球
duration_s	字符串	UFO 出现的可见时间	5 秒
sighting_en	text_en (英文文本)	目击的自由文本描述	Bright blue fireball in the distance during a rainstorm. Not a lightning storm, no thunder. No relevant towers or objects in the area.

表 9.1 有两点需要解释。首先，使用内置后缀将 Solr 字段名设为动态字段（第 5 章介绍过动态字段），例如字符串字段使用后缀 `_s`，日期字段使用后缀 `_dt`。其次，从原始数据中获得一些新的字段来实现分面。从每条目击数据的 `sighted_at_dt` 字段中提取月份信息映射到 `month_s` 字段，这让我们能够专注于一年中的月份信息。虽然这不是必须的，但它为用户提供了探索数据集的另一个角度。这样就能回答“某个特定的形状是否更频繁地出现在某些月中？”这样的问题。总体来说，这是设计数据模型来提供更好的用户体验的一个举例。

UFO 数据集中对目击情况有一两句话的简短描述，也有一些更为详细的描述。平均的描述长度约为 1100 个字符和 216 个词，还有数千条超过 3000 个字符的描述。因此，对这个数据集而言，Solr 的高亮是必备功能选项。现在准备来实现高亮吧！

9.2.4 开箱即用的高亮

Solr 预配置了高亮功能，不费吹灰之力就能实现搜索结果高亮。高亮是 Solr 的核心功能之一，也是几乎所有搜索应用必备的一个重要功能。在本节中，你将了解

如何启用高亮功能，以及如何控制搜索结果中每个文档生成的高亮片段数量。

## 基本高亮

代码清单 9.1 是图 9.1 中 Solr 的查询结果。<sup>1</sup>

代码清单 9.1 简单查询启用高亮功能

### 查询请求

```
http://localhost:8983/solr/ufo/select?q=blue fireball in the rain&
df=sighting_en&
wt=xml&
rows=10&
hl=true
```

← 设置该查询的默认搜索字段是 sighting\_en。

← 启用内置搜索结果高亮搜索组件，名为“highlight”。

### 搜索结果

```
<response>
  <lst name="responseHeader"> ... </lst>
  <result name="response" numFound="9687" start="0">
    <doc>
      <str name="id">20041130/.../2bbc6dc90efcbb8fb8f54ba23e07bd0a</str>
      ...
      <arr name="sighting_en">
        <str>Bright blue fireball in the distance during a rain storm.
          Not a lightning storm, no thunder. No relevant towers or
          objects in the area.</str>
      </arr>
      <long name="_version_">1431225103138422786</long>
    </doc>
  </result>
  <lst name="highlighting">
    <lst name="20041130/.../2bbc6dc90efcbb8fb8f54ba23e07bd0a">
      <arr name="sighting_en">
        <str>Bright <em>blue</em> <em>fireball</em>
          in the distance during a <em>rain</em> storm. Not a
          lightning storm, no thunder</str>
      </arr>
    </lst>
  </lst>
</response>
```

← 搜索结果中第一个文档的所有字段。

← 高亮结果返回在 XML 响应文档的一个单独部分里。

在搜索结果中高亮显示第一个文档。

← 第一个文档的高亮片段用 <em> 标记。

请注意，Solr 返回的高亮片段在一个单独的元素中：<lst name="highlighting">。因此，客户端应用程序必须处理这些信息，才能对其进行显示。搜索结果的第一个文档中没有 XML 转义字符的高亮片段如下<sup>2</sup>：

- <sup>1</sup> 第 4 章的查询清单工具可以在本地 Solr 服务器上执行这个请求：java -jar solr-in-action.jar listing 9.1。
- <sup>2</sup> Solr 返回的高亮字段中包含 XML 转义字符，例如，<用转义字符 <em> 表示。为提高可读性，本章使用粗体和阴影来显示高亮字段。

Bright **blue fireball** in the distance during a **rain** storm. Not a lightning storm, no thunder

除了额外的文字“Not a lightning storm, no thunder”之外,这个高亮片段与图 9.1 中显示的非常接近。在本章后面将介绍如何使用 `PostingsHighlighter` 组件得到图 9.1 中的确切结果。

现在用最简单的办法(设置查询参数 `hl=true`)来实现高亮效果。

正如 Solr 的大多数搜索组件一样,高亮组件支持多种配置参数的微调。下面通过使用几个可选的参数来了解一下处理过程。

### 每个结果生成多个高亮片段

在某些情况下,每个结果高亮显示一个片段,这可能还不足以为用户提供足够的上下文信息,让他判断这个结果是否值得进一步研究。例如,查询返回的第二个文档中高亮显示的文字如下:

. Horizontal, no arc. Fast. Huge. I described it to a friend who said it was a **"Blue Fireball**

乍看之下,这个结果很符合查询条件,因为它提到了 **Blue Fireball**。但好像这里还有更多的东西,特别是它在查询结果中被视为第二相关的文档。接下来,使用 `hl.snippets` 参数在该文档和其他文档中梳理出更多的上下文。代码清单 9.2 是 `hl.snippets=2` 时(允许每个文档最多高亮显示两个片段)的显示结果。

#### 代码清单 9.2 使用 `hl.snippets` 参数为每个文档生成更多片段

##### 查询请求

```
http://localhost:8983/solr/ufo/select?q=blue fireball in the rain&
df=sighting_en&
wt=xml&
hl=true&
hl.snippets=2
```

请求 Solr 为每个文档生成 2 个片段。

XML 响应文档的高亮部分。

##### 搜索结果

```
<lst name="highlighting">
  <lst name="20041130/20041204/lancaster/oh/fireball/
    2bbc6dc90efcbb8fb8f54ba23e07bd0a">
    <arr name="sighting_en">
      <str>Bright <em>blue</em> <em>fireball</em> in the distance during a <em>rain</em> storm. Not a
        lightning storm, no thunder</str>
    </arr>
```

搜索结果中第一个文档仅返回了一个片段。

```

</lst>
<lst name="20051110/20051116/lakeoswego/or/cylinder/
  4bae25d796ba82677ea0d77b36d08faf">
  <arr name="sighting_en">
    <str>. Horizontal, no arc. Fast. Huge. I described it to a friend
      who said it was a
        "<em><em><em>Blue</em></em></em> <em><em><em>Fireball</em></em></em></str>
    <str>Brilliant <em><em><em>blue</em></em></em> oblong object zooms
      horizontally across southern sky at 2 in the morning. I
        awoke
    </str>
  </arr>
</lst>
...
</lst>

```

搜索结果中第二个文档的第一个片段。

搜索结果中第二个文档的第二个片段。

下面是由第二个搜索结果生成的没有 XML 转义字符的两个片段：

- 1 Horizontal, no arc. Fast. Huge. I described it to a friend who said it was a **"Blue Fireball"**
- 2 Brilliant **blue** oblong object zooms horizontally across southern sky at 2 in the morning. I awoke

该文档中的第二个片段仍然没有提及 **rain**。结果中有趣的地方是，这些片段按相关度进行排序，与查询最接近的片段排在最前面。但事实上，在目击数据集中，第一个片段排在第二个片段后面。这里将此作为练习留给读者，读者可通过检查代码清单 9.2 执行返回的结果来验证这种情况。

正如第一个文档中所看到的高亮效果那样，设置 `hl.snippets=2` 并不能保证每个文档总能生成两个高亮片段。结果中只有第一个句子符合这个查询，因此 Solr 只返回了一个高亮片段。参数 `hl.snippets` 的值应被视为每个结果返回高亮片段的数量上限。

### 9.2.5 高亮具体细节

了解了高亮效果之后，接下来深入了解高亮背后发生的事情。第 4 章曾介绍过，高亮组件包含在搜索处理器的默认组件列表中。图 9.4 给出了高亮搜索组件在查询处理链中所处的位置。

`solrconfig.xml` 文件中定义了一个名为 `highlight` 的搜索组件。代码清单 9.3 给出了高亮搜索组件的简要定义。



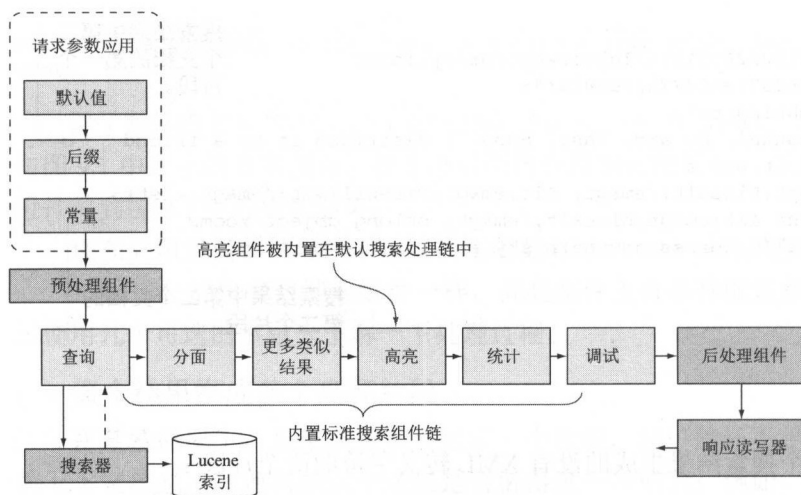


图 9.4 查询处理管道显示了高亮组件在搜索处理器组件链中所处的位置

## 代码清单 9.3 在 solrconfig.xml 中默认高亮搜索组件的 XML 定义

```

<searchComponent class="solr.HighlightComponent" name="highlight">
  <highlighting>
    <fragmenter name="gap" default="true"
      class="solr.highlight.GapFragmenter">
      ...
    </fragmenter>
    <fragmenter name="regex" class="solr.highlight.RegexFragmenter">
    ...
    </fragmenter>
    <formatter name="html" default="true"
      class="solr.highlight.HtmlFormatter">
      ...
    </formatter>
    <encoder name="html" class="solr.highlight.HtmlEncoder" />
    <fragListBuilder name="simple"
      class="solr.highlight.SimpleFragListBuilder"/>
    ...
    <fragmentsBuilder name="default" default="true"
      class="solr.highlight.ScoreOrderFragmentsBuilder">
      ...
    </fragmentsBuilder>
    ...
  </highlighting>
</searchComponent>

```

定义高亮搜索组件。

使用 GapFragmenter 作为默认片段。

使用正则表达式的替代片段。

使用 HTML 标记对词项进行高亮。

根据词向量，使用 FastVector-Highlighter 生成片段。

图 9.4 中需要注意一点，高亮组件位于查询处理链的下游。因此，高亮组件每次只能对一页搜索结果进行高亮处理。代码清单 9.1 中的查询示例请求了 10 个文档（rows=10），高亮组件在每次请求中就只能处理 10 个文档。因此，如果使用一个

较大的页面大小，比如 1000，那么每次请求中高亮组件的处理量很大，这会对查询响应时间造成负面影响。

通过设置 `hl.fl` 参数，高亮组件就能知道对文档的哪些字段进行高亮显示。如果像这里的示例没有设置 `hl.fl` 参数，那么 Solr 将回退到通过 `df` 参数设置的默认查询字段，示例中的默认查询字段是 `sighting_en`。对这个 UFO 搜索应用举例而言，唯一有意义的高亮显示字段是 `sighting_en`。使用多字段的一个例子是：数据包含 `title` 和 `body` 两个字段，并且打算在这两个字段上高亮显示查询词项。在这种情况下，需要设定 `hl.fl=title,body`，这样就能在 `title` 和 `body` 字段上都进行高亮显示。

对于 `hl.fl` 列表的每个字段，高亮组件将根据 `hl.snippets` 参数来决定生成多少个高亮片段，该参数的默认值为 1。这个参数设定了每个字段生成的高亮片段的数量上限。如果设定 `hl.snippets=2`，那么 Solr 在每个字段上可能会生成 0 个、1 个或 2 个高亮片段。这个参数在前一节的示例中出现过。

这里的高亮组件处理一小部分文档 (`rows=10`)，它知道在哪些字段上生成高亮片段 (`df=sighting_en`)，并且知道在每个字段上生成多少个高亮片段 (`hl.snippets=2`)。下一步是对高亮显示的文本进行重新分析。图 9.5 说明了高亮组件如何生成高亮片段。

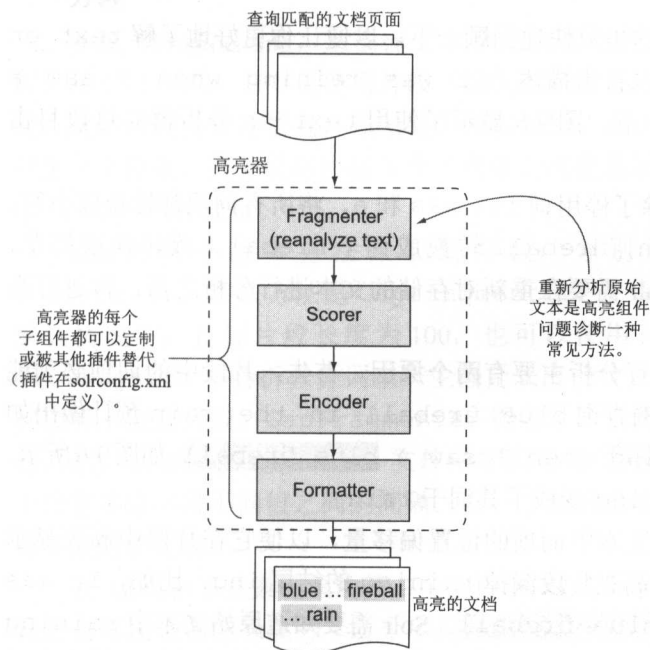


图 9.5 Solr 高亮组件的主要组成

## 文本分析

在 Solr 高亮显示结果之前，它需要访问原始文本。正如第 5 章所学到的，Solr 需要访问原始文本中的字段并将其存储起来。在这个示例中，高亮显示 sighting\_en 字段。这是一个动态字段，在 schema.xml 文件中的声明如下：

```
<dynamicField name="*_en" type="text_en"
  indexed="true" stored="true" multiValued="true" />
```

当 Solr 得到该字段的原始文本之后，它需要使用配置好的索引阶段分析器进行重新分析。对于 sighting\_en 字段，分析器通过 text\_en 字段类型进行配置，具体参见 schema.xml 文件中的完整定义：

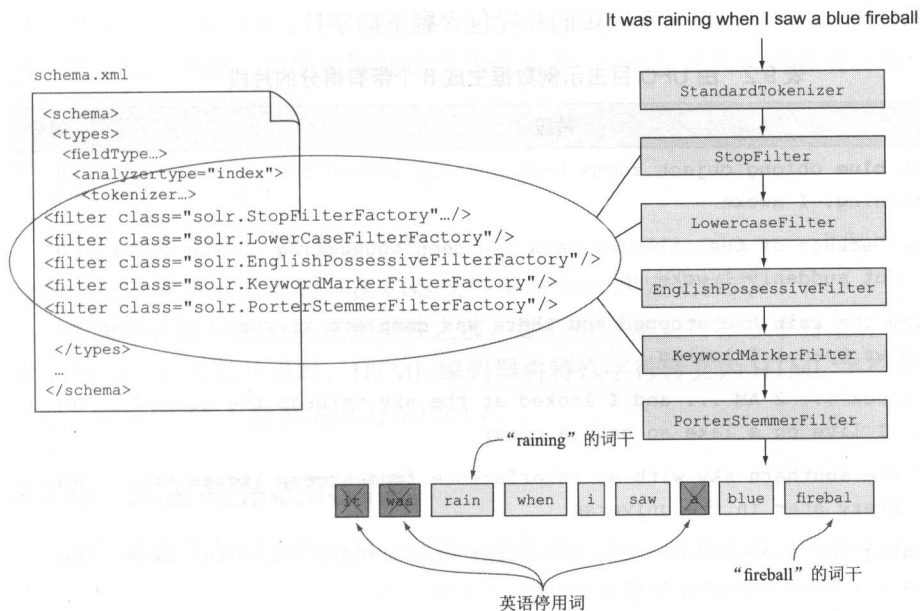
```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" .../>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory" .../>
    <filter class="solr.PorterStemFilterFactory"/>
  </analyzer>
  ...
</fieldType>
```

第 6 章介绍过文本分析，这里只快速回顾一下，以便让你更好地了解 text\_en 分析器执行的文本转换。我们以目击描述 “It was raining when I saw a blue fireball” 为例来看一下。图 9.6 显示了使用 text\_en 分析器对这段目击描述文本进行的转换。

从图 9.6 中看出，Solr 删除了停用词 it、was 和 a，将所有词项都转换成小写，将 raining 转换成词干 rain，fireball 转换成词干 firebal。该转换很简单，但你可能会有疑问，为什么 Solr 需要在重新对存储的文本进行分析之后，再进行高亮显示呢？

对文档的原始文本重新进行分析主要有两个原因。首先，片段中的词项必须能与查询词项相比较。Solr 对示例查询 blue fireball in the rain 预计做出如下的高亮效果：It was **raining** when I saw a **blue fireball**。如图 9.6 所示，在高亮前分析原始文本，raining 变成了其词干 rain。

其次，Solr 需要知道原始文本中词项的位置偏移量，以便它在片段中高亮显示词项。我们不希望 Solr 生成的高亮片段漏掉 raining 的结尾 ing，比如，It was **raining** when I saw a **blue fireball**。Solr 需要知道原始文本中 raining 开始和结束的位置。位置偏移量很重要，通过它 Solr 才能知道什么时候该高亮显示一个短语，而不是显示短语的一部分。

图 9.6 通过 `text_en` 分析器对虚拟的目击描述文本进行文本分析

## 分段

分段 (Fragmenting) 是选择文本字段中零个或多个子片段进行高亮显示的过程。基于这一点，书中直使用片段 (snippet) 这个词，其表示已排名和格式化的片段，这些片段得分足够高，在结果中得以返回。Solr 把文本分割成片段，每个片段产生一个分数，得分最高的前  $N$  个片段作为高亮显示片段被返回，这里的  $N$  由参数 `hl.snippets` 设定。

Solr 有两个基本方法来实现分段，分别是 `GapFragmenter` 和 `Regex Fragmenter`。默认方法是 `GapFragmenter`，它基于一个目标长度来选择片段。默认情况下，目标片段长度为 100，也可以使用 `hl.fragsize` 参数进行修改。`hl.fragsize` 不会拆分分词来产生固定长度的片段，而 `GapFragmenter` 会在分词边界上创建片段。`GapFragmenter` 得名于，它避免了在跨越较大位置间隔上创建片段。例如，在 `schema.xml` 文件中字段定义的 `positionIncrementGap` 属性上设置多值字段不同值之间的间隔。下一节介绍通过 `GapFragmenter` 参数设置多值字段的值间隔。

表 9.2 显示了搜索结果的第二个文档中，通过 `GapFragmenter` 生成的 8 个片段。该目击描述数据的长度为 742 个字符，生成 8 个片段有一定道理。前 7 个片段大约

每个片段各 100 个字符，第 8 个较短的片段包含剩下的字符。

表 9.2 由 UFO 目击示例数据生成 8 个带有得分的片段

片段	得分
Brilliant <b>blue</b> oblong object zooms horizontally across southern sky at 2 in the morning. I awoke	1.0
suddenly because of the silence. <b>Rain</b> had been thundering on the glass ceiling, but suddenly I woke up	1.0
realizing the <b>rain</b> had stopped and there was complete silence (I sleep with windows open.) I looked	1.0
at the clock ... 2 AM ... and I looked at the sky through the glass ceiling. (I live on a lake so have a clear	0.0
view of the southern sky with no interference from trees, lights or houses.) Every star in the universe	0.0
was shining and suddenly, across the horizon, zoomed a brilliant <b>blue</b> something ... from west to east	1.0
. Horizontal, no arc. Fast. Huge. I described it to a friend who said it was a " <b>Blue Fireball</b>	2.0
." It was a beautiful experience.	0.0

Solr 还提供了 `RegexFragmenter` 方法，通过正则表达式在文本中选择片段。例如，如果要在英文文本中生成句子来作为片段，可以使用诸如 `[-\w ,/\n\"'] {20,200}` 这样的正则表达式。这也是 `solrconfig.xml` 文件中的预配置模式。

评分

分段组件使用了评分组件的子组件为片段评分。默认评分组件的实现方法 (`QueryScorer`) 会计算每个分段中出现了多少个查询词项。与示例查询匹配的第三个文档的下面这个片段得分为 3.0。

Bright **blue fireball** in the distance during a **rain** storm. Not a lightning storm, no thunder

本章后面介绍 `PostingsHighlighter` 组件时，我们会学习另一种评分组件，它采用了更先进的评分算法。

编码与格式

`hl.formatter` 参数指定了一个格式化组件，用来对词项进行高亮显示。默认格式化组件 (`hl.formatter=simple`) 将每个词项包含在任意一段文本中。这种方法适合适用 `HTML` 标签来高亮显示查询词项。默认情况下，`simple` 格式化组件将

词项放在 `<em>` 标签中, 也可以使用 `hl.simple.pre` 和 `hl.simple.post` 参数覆盖默认值。例如, 如果想要使用层叠样式表 (CSS) 对高亮词项进行格式化, 在查询中添加以下参数即可:

```
hl.simple.pre=<span class="foo">&
hl.simple.post=</span>
```

使用这些参数, 高亮显示的词项显示为 `<span class="foo">term</span>` 这样的形式。然后, 在 CSS 文件中对 `foo` 类定义样式。

在每个片段传递给格式化组件之前, 编码组件需要对特殊字符进行编码。当生成一个 HTML 高亮片段时, HTML 编码器将特殊字符转义为 HTML 字符实体引用; 例如, 双引号 (") 会被编码为 `&quot;`。

## 9.2.6 改善高亮显示结果

以上介绍了 Solr 中确定每个文档中的哪些片段应该高亮显示的基本过程。接下来, 让我们通过更多例子进一步了解如何改善高亮显示结果。

### 与分面一起使用

第 8 章介绍过, 分面允许用户细化搜索条件对数据集进行探索。UFO 数据集里有三个字段可以生成有意义的分面, 这三个字段分别为: `shape`、`location` 和 `month`。`month` 字段是为支持分面特意从 `sighting` 日期字段中提取出来的。代码清单 9.4 是生成分面并高亮显示的查询表达式。

代码清单 9.4 启用分面与高亮的查询示例

```
http://localhost:8983/solr/ufo/select?q=blue fireball in the rain&
df=sighting_en&
wt=xml&
hl=true&
hl.snippets=2&
hl.fl=sighting_en&
facet=true&
facet.limit=4&
facet.field=shape_s&
facet.field=location_s&
facet.field=month_s
```

每个分面  
字段最多  
生成 4 个  
类目。

启用高亮。

每个文档产生  
两个片段。

高亮 sighting\_en  
字段。

启用分面。

为 shape\_s、location\_s  
和 month\_s 字段使用分  
面。

图 9.7 显示了每个分面的前 4 个值, 其结果由代码清单 9.4 的查询表达式生成。

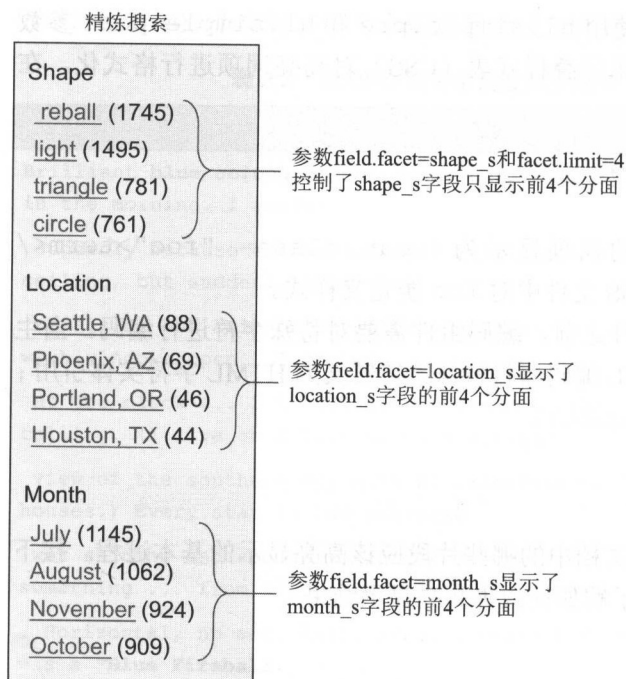


图 9.7 shapw、location 和 month 分面的前 4 个值，这是示例查询 blue fireball in the rain 的搜索结果

这些结果很清晰地显示了“Seattle, WA”是目击 UFO 最频繁的地点，light 是最频繁被报道的 UFO 形状。试想，如果用户在 shape 分面下选择 light 会发生什么，该查询表达式将包括筛选查询 `fq=shape_s:light`。使用过滤器的分面查询在 8.6 节介绍过，本节主要关注如何在高亮中使用分面。

由于用户提供了所要寻找的很多信息，所以应该将分面信息纳入到高亮显示的信息中。特别是在选择最佳片段进行高亮显示时，也应该考虑词项 light。但使用过滤器时，由于 light 一词出现在过滤查询部分 (fq) 而不在查询条件的 q 部分，高亮组件不会将额外的词项 light 纳入到选择的片段中，或对其高亮显示。因此，我们需要一种方法来控制高亮显示的查询词项，使用该方法不要对匹配文档的查询表达式进行修改。

这里使用一个稍作修改的查询来驱动高亮并用来匹配文档，这是 Solr 中常见的用法。Solr 使用自定义参数 `hl.q` 支持这一功能。代码清单 9.5 显示了如何使用参数 `hl.q` 将词项 **light** 添加到使用高亮的查询中。



代码清单 9.5 使用参数将分面词项添加到高亮结果中

```
http://localhost:8983/solr/ufo/select?q=blue fireball in the rain&
df=sighting_en&
wt=xml&
hl=true&
hl.snippets=2&
hl.fl=sighting_en&
hl.q=blue fireball in the rain light&
fq=shape_s:light
```

在高亮查询中也包含分面词项“light”。

使用光的形状分面进行过滤。

使用 light 分面过滤器将词项 light 添加到 hl.q 参数值之后, Solr 会返回如下最前面的两个片段。相比不将词项 light 包含在高亮结果中而言, 该返回结果更加人性化:

- 1 and orange as well as **blue light** and what appeared to be a **fireball**. After that no more **lights** and no more
- 2 Neon **Blue** Flashes of **light** around Perry Ohio Nuclear Power PLant **light** up night sky and cause power

### 高亮显示短语

本章一直使用的示例查询 blue fireball in the rain 是很宽松的, 只要文档包含词项 blue、fireball 或 rain, 不管词项处于文档中的什么位置, 该查询都能匹配出这篇文档。如果启用查询调试 (debug=true), 可以看到被执行的 Lucene 查询表达式 (查询文本分析之后的):

```
sighting_en:blue OR sighting_en:firebal OR sighting_en:rain
```

这个查询匹配出 9706 个文档。让我们考虑更严格的查询, 只关心在 sighting\_en 字段中确切出现 blue fireball 这个短语的文档。这时底层的 Lucene 查询表达式稍有不同: sighting\_en:"blue firebal" AND sighting\_en:rain。该查询的约束较强, 只匹配出 2 个文档。

重要的一点是, Solr 能正确地高亮显示短语 "blue fireball", 而不是单独高亮显示词项 blue 或 fireball。例如, 在下面这个虚拟的片段中第二个 fireball 不会高亮显示:

```
I saw a blue fireball in the sky, it was an awesome fireball.
```

在后台, hl.usePhraseHighlighter 参数控制这个行为, 默认情况下它是启用状态 (true)。如果设置 hl.usePhraseHighlighter=false, 那么 Solr 可能返回单独高亮显示词项 blue 或 fireball 的片段。关键之处在于, Solr 在高亮显示

查询短语时处理得当，归功于 `hl.usePhraseHighlighter` 参数在 Solr 4 中默认启用。

### 高亮显示多值字段

Solr 也能对多值字段高亮显示。试想一下，有一个多值字段记录了目击时刻靠近 UFO 的其他物体的简短描述。这是完全虚构的，并不是 UFO 目击数据集的一部分，示例中的索引构建程序创建了带有 `_objects_en` 字段的文档：

```
<arr name="nearby_objects_en">
  <str>arc of red fire</str>
  <str>cluster of dark clouds</str>
  <str>thunder and lightning</str>
</arr>
```

代码清单 9.6 显示了匹配 `nearby_objects_en` 字段并高亮显示的查询表达式。

#### 代码清单 9.6 高亮多值字段的查询

```
http://localhost:8983/solr/ufo/select?q=fire cluster clouds thunder&
df=nearby_objects_en&
wt=xml&
hl=true&
hl.snippets=2
```

← 匹配与高亮假设的多值字段

查询 `fire cluster clouds thunder`，可能会得到诸如 `fire cluster of dark clouds thunder` 这样的片段。Solr 关心的是不产生跨越多值字段不同值的片段。对于这个查询，Solr 正确地生成了两个高亮显示的片段：

- **cluster** of dark **clouds**
- arc of red **fire**

之所以得到这样的效果，是因为 `GapFragmenter` 参数支持 Solr 的 `positionIncrementGap` 属性设置了多值间隔。对于 `text_en` 字段类型，`positionIncrementGap` 的值设为 100。如果想要高亮显示多值字段的所有值，设置 `hl.preserveMulti=true`。在这种情况下，Solr 将返回这个示例文档中的所有三个值。

### 高亮参数小结

在学习 Solr 其他高亮实现方法之前，我们先总结一下 Solr 中的默认高亮组件。表 9.3 介绍了默认高亮组件的一些参数，其中包括本章没有介绍到的一些参数，它们大多数一看就懂。

### 表 9.3 适用于默认高亮组件的最常见参数

参数	说明	默认值
hl	为查询启用高亮功能，设置为 true	false
hl.snippets	每个字段生成的最大片段数	1
hl.fl	一个或多个字段列表，用逗号分隔，以生成高亮片段	<no default> 无默认值则回头使用 df 字段
hl.fragmenter	指定要使用的分段器组件，在 solrconfig.xml 中定义分段器	gap
hl.fragsize	设置每个片段的目标长度，无严格上限	100
hl.q	与原始查询 (q) 相比，高亮用途的替代查询能够高亮不同的词项，灵活性更好	<no default>
hl.alternateField	当一个字段没有可高亮的片段时，指定一个存储字段用于显示	<no default>
hl.formatter	指定要使用的格式化组件，在 solrconfig.xml 中定义格式化组件	simple
hl.simple.pre	在每个高亮词项前添加任意文本	<em>
hl.simple.post	在每个高亮词项后添加任意文本	</em>
hl.requireFieldMatch	当高亮多个字段，为实现高亮，3 需要一个字段与查询相匹配	false
hl.maxAnalyzedChars	对非常大的字段进行分段时，用于指定最大的分析字符数	51200
hl.usePhraseHighlighter	若启用该参数，Solr 仅高亮匹配到的短语。短语中的单个词项在文本其他位置出现时不高亮	true
hl.mergeContiguous	设置为 true 的话，Solr 会将相邻的片段合并为一个片段	false (向后兼容)
hl.highlightMultiTerm	设置为 true 的话，为区间 / 通配符 / 模糊 / 前缀查询启用高亮	false
hl.preserveMulti	设置为 true 的话，在多值字段的所有值上执行高亮，并保留多值字段的值顺序	false

## 字段级覆写

很多高亮组件的参数都可以进行字段级覆写。例如，搜索应用有 `title` 和 `body` 两个字段。标题总是很短，因而只要求产生一个片段；但你可能想在较长的 `body` 字段上生成三个片段，只需要设置 `f.body.hl.snippets=3` 来覆写 `body` 字段的默认值 1 即可。一般情况下，在参数前添加 `f.<fieldname>.` 对任意参数应用字段级覆写，如 `f.body.hl.snippets`。

至此，你应该对 Solr 默认高亮组件及如何使用可选参数来改善高亮效果有了充分了解。下面介绍一些弥补默认高亮组件缺点的其他高亮组件。

## 9.3 使用FastVectorHighlighter组件提升性能

默认高亮组件最常见的问题是，大文本字段的高亮显示处理速度太慢。造成缓慢的主要原因是，它在查询时需要重新分析原始文本。对于大文本字段或文本分析很复杂时，处理速度会很慢。为解决这个问题，Solr 提供了 FastVectorHighlighter 组件。因为在生成片段时，它跳过了分析步骤，因此它比默认高亮组件速度更快。

### 默认高亮组件的性能

需要说明的是，当默认的高亮器执行速度过于缓慢时，这会导致无法在 UFO 目击数据集中找到结果。在非正式的基准测试中，发送成千上万条随机生成的查询时，默认高亮组件总是非常快，即使页面大小为 50 也是如此。如果有很多更大的文档或在很多字段上处理高亮，处理速度可能会受到影响。虽然在默认高亮组件处理速度太慢时，FastVectorHighlighter 组件能有所帮助，但这并不意味着，默认高亮组件的处理速度会因为数据而变得很慢。

FastVectorHighlighter 组件需要知道词项位置和偏移量信息来实现高亮，因此它需要使用索引构建过程中的计算信息。要使用 FastVectorHighlighter 组件，需要高亮显示的所有字段都必须进行索引，并启用 termVectors、termPositions 和 termOffsets 参数。为了在 sighting\_en 字段上使用 FastVectorHighlighter 组件，对它定义如下：

```
<field name="sighting_en" type="text_en" indexed="true" stored="true"
  termVectors="true" termPositions="true" termOffsets="true" />
```

这看起来似乎很容易，从中收获了什么呢？对于初学者来说，要使用这些设置，你需要重新索引所有文档。更大的一个问题是，这些属性会大大增加索引的大小，对于本章较小的 UFO 目击数据集而言，索引大小会从 69 MB 增加到 109 MB。索引大小的增加对小数据集没有太大的问题，但数据规模一旦过大，这种增长就可能成为一个问题。对词向量、位置信息、偏移量信息进行存储也（稍微）减缓了索引构建速度，这在近实时搜索的高吞吐量环境中可能也会是一个问题。

要使用 FastVectorHighlighter 组件，就需要重新索引所有文档。在这个示例中需要做的是，在 schema.xml 文件中为字段 sighting\_en 添加新的字段定义（参见上一节），重启 Solr 服务器，并重新运行 IndexUfoSightings 程序，操作如下：

```
java -jar solr-in-action.jar ufo -jsonInput
  ➡ FULL_PATH_TO_UFO_SIGHTINGS_DIR/ufo_awesome.json
```

如代码清单 9.7 所示, 在查询表达式中设置 `hl.useFastVectorHighlighter = true`, 来启用 `FastVectorHighlighter` 组件。

#### 代码清单 9.7 启用 `FastVectorHighlighter`

```
http://localhost:8983/solr/ufo/select?q=blue fireball in the rain&
df=sighting_en&
wt=xml&
hl=true&
hl.snippets=2&
hl.useFastVectorHighlighter=true
```

必须显式启用  
`FastVectorHighlighter`。

`FastVectorHighlighter` 组件与默认高亮组件的高亮结果具有一定可比性。在性能方面, 只有在文本分析很复杂或在大文本字段上处理高亮显示时, 才能感受到性能的提升。不过这里的 UFO 数据集没能提供足够的复杂性, 让 `FastVectorHighlighter` 组件发挥出的性能优势, 本节介绍 `FastVectorHighlighter` 组件的目的是, 如果默认高亮组件在搜索应用中出现性能问题, 可以考虑使用这种方式来替代默认高亮组件。

## 9.4 PostingsHighlighter组件

本章最后介绍一下 Solr 最新的高亮实现方法——`PostingsHighlighter` 组件, 它比默认高亮组件的处理速度快, 且无须 `FastVectorHighlighter` 组件的词向量开销。`PostingsHighlighter` 的得名是因为, 它依赖倒排索引中倒排信息的词项偏移量。相比之下, `FastVectorHighlighter` 组件需要在索引中有一个单独的数据结构, 用于检索词项位置和偏移量, 而 `PostingsHighlighter` 组件可以从倒排列表中直接访问这些信息, 这是 Lucene 4 的一个新功能。回忆一下 Lucene 的倒排索引, 倒排列表包含一组文档中的词项在每个文档中出现的位置及词频的信息。在 Lucene 4.0 版本中, 也有在 `posting` 列表中存储词项位置和偏移量的选项。

要使用 `PostingsHighlighter` 组件, 在字段上必须设置 `storeOffsetsWithPositions="true"`, 然后索引该字段。对于本节的示例, 需要在 `schema.xml` 文件中添加 `sighting_en` 字段定义:

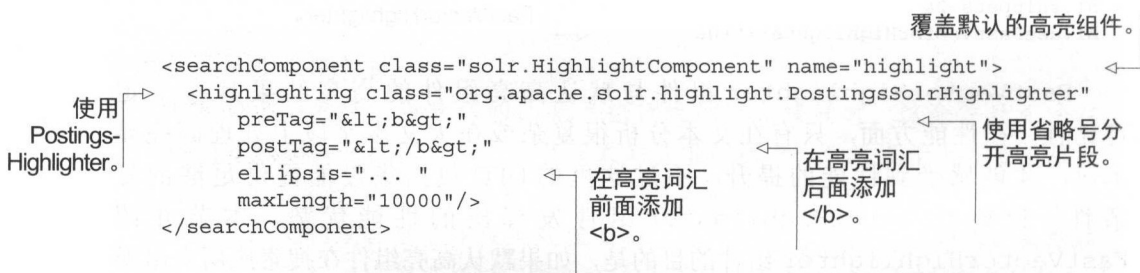
```
<field name="sighting_en" type="text_en" indexed="true" stored="true"
  storeOffsetsWithPositions="true" />
```

之前通过字段名称中的 `_en` 后缀定义了一个动态字段。因为需要添加 `storeOffsetsWithPositions` 参数, 所以需要在 `schema.xml` 文件中显式定义

sighting\_en 字段。在添加字段定义后,需要重启 Solr 服务器并重新索引所有文档。9.3 节曾介绍过,使用 FastVectorHighlighter 组件时,对词向量、位置信息和偏移量信息进行存储,导致索引大小从 69 MB 增加到 109 MB。在倒排列表中存储偏移量和位置信息,索引大小会从 69 MB 增加到 85 MB,这对大型索引而言是有显著差异的。

与 FastVectorHighlighter 组件不同的是,这里还需要在 solrconfig.xml 文件中使用代码清单 9.8 的 XML 定义来配置 PostingsHighlighter 组件。

代码清单 9.8 在 solrconfig.xml 配置 PostingsHighlighter



请注意,必须在 solrconfig.xml 文件中已有的高亮组件定义之后或删除默认高亮组件定义,才能添加代码清单 9.8 的 XML 定义。应用更改后的配置,然后重新执行代码清单 9.2 中的查询,使用 http 工具查看 PostingsHighlighter 组件的高亮效果:

```
java -jar solr-in-action.jar listing 9.2
```

使用 PostingsHighlighter 组件返回的高亮片段与使用默认高亮组件返回的结果类似,第一个文档中生成了如下高亮片段:

Bright blue fireball in the distance during a rain storm.

请注意,PostingsHighlighter 组件使用 Java 的 BreakIterator 类(参见 java.text.BreakIterator)的一个句子识别方法来进行分段的,因此它没有返回默认高亮组件返回的额外词项(Not a lightning storm, no thunder)。对于代码清单 9.2 的查询返回结果中的第二个文档,PostingsHighlighter 组件返回如下结果:

Brilliant blue oblong object zooms horizontally across southern sky at 2 in the morning. ... Rain had been thundering on the glass ceiling, but suddenly I woke up realizing the rain had stopped and there was complete silence (I sleep with windows open.)



与其他高亮组件不同的是，PostingsHighlighter 组件返回的所有片段都出现了一个以省略号 (...) 分隔的连续字符串。

除了在速度和减少索引开销上有优势，PostingsHighlighter 组件还为片段评分采用了一个更先进的相似度计算方法——BM25。与默认高亮组件计算每个片段中查询词项的词频不同，BM25 是目前高级的计算文档或片段与查询之间相似度的 tf-idf 评分函数。<sup>3</sup> BM25 评分函数提升了词项出现频率小的索引片段的权重。在返回结果的第二个文档中我们能看到一些依据，其中词项 rain 被高亮显示，因为它在索引中出现的词频比 blue 或 fireball 少，所以 BM25 评分函数赋予它更高的权重。

PostingsHighlighter 组件的主要缺点是，它要求在索引构建过程中设置精确的词项偏移量。我们使用第 6 章接触过的 Solr 分析表单来看看如何在文本分析阶段计算词项的偏移量。图 9.8 显示了本章示例如何在文本分析阶段计算词项偏移量。

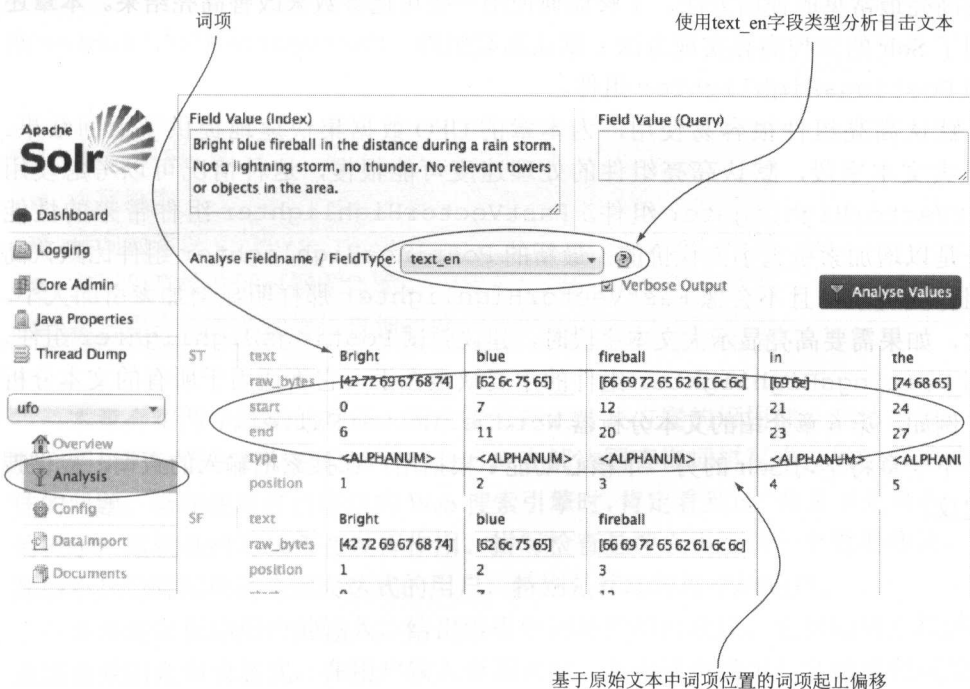


图 9.8 文本分析过程中计算词项开始及结束偏移量的分析表单截图

从图 9.8 中可以看到，text\_en 字段上使用 StandardTokenizer 组件来计算词项开始及结束的偏移位置。例如，如果从 0 的位置开始计数，则词项 fireball

<sup>3</sup> BM25 背后的数学原理超出本书范围，更多信息请参考维基百科中的 BM25 介绍页面 ([http://en.wikipedia.org/wiki/Okapi\\_BM25](http://en.wikipedia.org/wiki/Okapi_BM25))。



的开始位置为 12，结束位置为 20。但也有一些分词器和分词过滤器不能准确地反馈偏移量<sup>4</sup>。因此，PostingsHighlighter 组件并不是任何情况都适用，这取决于文本分析需求。

## 9.5 本章小结

本章实现了一个寻找 UFO 目击描述的基础搜索应用。具体来说，本章先设计了一个客户端应用程序对 Infochimps 提供的公开数据集（UFO 目击数据集）进行预处理并构建索引，然后介绍了如何对查询词项匹配的短片段进行高亮显示。这些片段根据查询相似度，使用内置的高亮搜索组件动态生成。为每个搜索结果选择最佳片段进行高亮显示，这是高亮组件的核心功能。

Solr 的高亮功能实现中有很多可选参数，这会让 Solr 新手感到困惑。本章主要关注高亮最常见的使用方法，了解如何使用一些可选参数来改善高亮结果。本章还介绍了 Solr 的三种高亮实现方法：默认高亮组件、FastVectorHighlighter 组件和 PostingsHighlighter 组件。

默认高亮组件很容易使用，为本章的 UFO 数据集快速地提供了返回结果。对于大文本字段，默认高亮组件的处理速度可能很慢，这种情况可以考虑使用 FastVectorHighlighter 组件。FastVectorHighlighter 组件带来的性能提升是以增加索引大小为代价的。最新的 PostingsHighlighter 组件比默认高亮组件更有效，且不会像 FastVectorHighlighter 那样明显增加索引的大小。因此，如果需要高亮显示大文本字段时，建议尝试 PostingsHighlighter 组件。使用 PostingsHighlighter 组件的主要缺点在于，它不适用于所有的文本分析器，例如，第 6 章介绍的文本分析器 WordDelimiterFilter。

下一章将学习 Solr 的另一个核心功能：根据用户在搜索框输入的查询，提供搜索建议。

---

4 对于词素处理器的问题更新列表请参见 <https://issues.apache.org/jira/browse/LUCENE-4641>。

# 10

## 查询建议

### 本章要点

- 拼写检查
- 查询词建议
- 实现预测性的即时搜索体验
- 基于用户历史活动提供查询建议

本章介绍两个简单的功能，用来提高搜索解决方案的可用性。具体来说，本章将介绍拼写检查和自动建议功能。拼写检查解决因查询词拼写错误导致搜索结果不佳的问题。当你使用自己喜欢的 Web 搜索引擎时，肯定看到过“你是不是要找……？”的提示，这就是拼写检查在发挥作用。拼写检查是搜索引擎的一个重要特征，帮助那些不想花时间构造查询表达式的用户，特别是移动环境中的用户。

自动建议根据用户的输入，给出索引中词项的即时建议。它帮助用户较快地构造出有效的查询表达式。在用户输入查询词时，从索引中找出与之相关的词项，即时提供反馈查询建议。本章首先介绍 Solr 的拼写检查搜索组件的工作原理，它是实现自动建议的基础。

## 10.1 拼写检查

本节将学习如何使用 Solr 的拼写检查搜索组件。自动拼写检查是一个核心搜索功能，让大多数用户不用思考就能轻松使用它。拼写检查的使用有 4 种常见情况，需要有所了解：

1. 查询包括一个或多个拼写错误的词，导致结果中得到不相关的内容。如果查询建议可用，搜索引擎应自动执行查询建议，向用户显示一条消息，例如，“显示的是 atmosphere”的搜索结果，或“仍然搜索 atmosphere”。
2. 查询包括罕见词，没有返回什么搜索结果。与此同时，存在可用的查询建议，并且能够得到多一些搜索结果。在这种情况下，搜索引擎提示用户“你是不是要找……?”。
3. 查询包括拼写正确的字词。虽然存在可用的查询建议，但两者的搜索结果情况差不多。在这种情况下，搜索引擎无须向用户提供建议。
4. 查询包括索引中不存在的词项，没有可用的查询建议。

从这 4 种情况可以得出拼写检查使用的两个关键要求。首先，需要一种方法识别出查询中每个词项的建议词。也就是说，在某种字典中查找与用户输入查询词相类似的词项。其次，需要知道每个建议词匹配了多少文档，这会对是否给出查询建议以及如何提示用户起到一定参考作用。本节介绍 Solr 如何处理这两方面。

为了本章内容的讲解需要，本节构建一个类似维基百科的在线百科全书索引。具体操作上，本节只使用维基百科英文文章的一小部分样例来填充索引。维基百科涉及各类主题，作为查询建议的基础。同时，维基百科的数据质量很高，在一定程度上可以保证所给查询建议的质量。具体来说，我们从维基百科的较新 XML 转储文件<sup>1</sup>中随机选择了 13 000 篇文章。由于维基百科全部数据超过了 40GB，需要几个小时才能下载、解压缩和索引，因此本节只使用其中一小部分数据来做实验。

### 10.1.1 索引维基百科的文章

首先要将维基百科的文章导入 Solr 中。为了简化这一过程，我们事先配置好一个名为 solrpedia 的 Solr 内核，位于 `$SOLR_IN_ACTION/example-docs/ch10/cores/solrpedia/` 目录。要构建该内核以及本章其他示例内核，执行以下命令：

```
cd $SOLR_IN_ACTION/example-docs/  
cp -r ch10/cores/ $SOLR_INSTALL/example/solr/
```

---

<sup>1</sup> 数据来源 <http://dumps.wikimedia.org/enwiki/latest/>。

启动本地工作站上的 Solr 示例服务器。如果它已经在运行，将其停止并重新启动：

```
cd $SOLR_INSTALL/example/  
java -jar start.jar
```

接下来，如图 10.1 所示，在 Solr 管理控制台中确认 solrpedia 内核的存在。



图 10.1 确认在 Solr 管理控制台可获得预配置的 solrpedia 内核

事先配置好的 solrpedia 内核使用 Solr 的数据导入处理器（DIH）从维基百科的 XML 数据转储文件中导入样例文章。本章的重点是拼写检查和自动建议，本节通过预定义的 DIH 配置导入数据。如果想了解如何配置 DIH 来导入维基百科的文章，请参见附录 C。

在数据导入之前，需要将 SOLR\_IN\_ACTION/example-docs/ch10/documents/solrpedia.xml 文件复制到 \$SOLR\_INSTALL/example/ 目录下，让 DIH 可以找到 XML 文档进行导入。如果在导入前你对数据感到好奇，可以使用任何一个文本编辑器查看 solrpedia.xml 文件。

复制 solrpedia.xml 文件之后，执行数据导入处理程序来构建 solrpedia 的索引。刷新 Solr 管理控制页面，在 solrpedia 内核下选择数据导入页。如图 10.2 所示填写表单，然后点击执行（Execute）按钮。

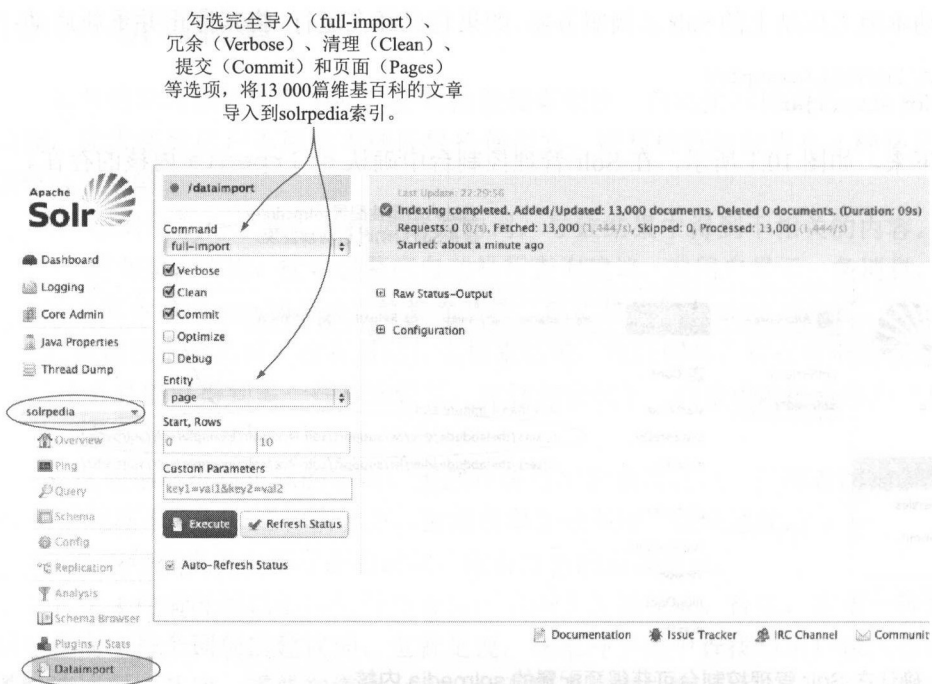


图 10.2 使用数据导入页从 solrpedia.xml 文件中将维基百科的文章导入到 solrpedia 内核

此时，solrpedia 的索引应该包含 13 000 个文档。通过使用查询表单，执行匹配所有文档的查询 (\*:\*) 来检查导入情况。现在，数据已经准备好了，接下来看看如何使用 Solr 的拼写检查功能。

10.1.2 拼写检查举例

使用 http 命令行工具执行代码清单 10.1，在 Solr 上查询拼写错误的词 “atmosphear”。

代码清单 10.1 在 Solr 上查询拼写错误的词 "atmosphear"

查询请求

http://localhost:8983/solr/solrpedia/select?q=atmosphear ← 拼写错误的查询。

搜索结果

```
<response>
  <lst name="responseHeader">...</lst>
  <result name="response" numFound="0" start="0"></result>
  <lst name="spellcheck">
    <lst name="suggestions">
      <lst name="atmosphear">
```

没有找到相关文档。

```

<int name="numFound">1</int>
...
<arr name="suggestion">
  <str>atmosphere</str>
</arr>
</lst>
<lst name="collation">
  <str name="collationQuery">atmosphere</str>
  <int name="hits">86</int>
  <lst name="misspellingsAndCorrections">
    <str name="atmosphear">atmosphere</str>
  </lst>
</lst>
</lst>
</response>

```

拼写检查组件建议“atmosphere”。

建议词在语料库中有 86 条搜索结果。

另外，还可以使用 `java -jar solr-in-action.jar listing 10.1` 执行该查询。请注意，对于拼写错误的查询词“atmosphear”，Solr 建议将词“**atmosphere**”作为替代词。搜索结果中还包含一个 `collation` 部分，用来说明建议词出现在多少个文档中，这里是 86 条搜索结果。搜索引擎对建议信息进行判断，进而确认是否符合之前讨论过的第一种情况。因此，在后台自动执行 `collationQuery`，给用户显示一条消息说明“显示的是 `atmosphere` 的搜索结果”，或“仍然搜索 `atmosphear`”。另外，还可以提示用户“你是不是要找 `atmosphere`?”。因为用户的原始查询没有返回搜索结果，而 `collationQuery` 返回很多搜索结果，所以假设用户是想查询 `atmosphere`，并在后台自动执行 `collationQuery`，这种做法是安全的。

尝试搜索名为 `anaconda` 的文章标题，如代码清单 10.2 所示。Solr 无法找到任何文档或给出查询建议。

#### 代码清单 10.2 在 `title` 字段搜索不存在的词项

##### 查询请求

```
http://localhost:8983/solr/solrpedia/select?q=title:anaconda
```

##### 搜索结果

```

<response>
  <lst name="responseHeader">...</lst>
  <result name="response" numFound="0" start="0"></result>
  <lst name="spellcheck">
    <lst name="suggestions"/>
  </lst>
</response>

```

没有对 `anaconda` 的搜索建议。

当用户对索引中不存在的词项进行搜索时，代码清单 10.2 是预料之内的搜索结果。在这种情况下，Solr 找不到相关文档，也无法做出有用的查询建议。此时，最

好是对那些频繁进行查询却没有查询结果或查询建议的情况进行日志分析，将其添加到一个外部词典中。拼写检查组件基于索引中的词项生成查询建议，下一节介绍这个组件的更多内容。

我们来看一个不应该使用查询建议的例子。代码清单 10.3 是查询 Julius 的结果，这个词的拼写是正确的，但 Solr 仍然返回了一些查询建议。

代码清单 10.3 忽略 Solr 建议的查询示例

查询请求

```
http://localhost:8983/solr/solrpedia/select?q=Julius&
df=suggest&
fl=title
```

使用建议字段作为默认搜索字段。

搜索结果

```
<response>
<lst name="responseHeader">...</lst>
<result name="response" numFound="4" start="0">
  <doc><str name="title">Julius Wegscheider</str></doc>
  ...
</result>
<lst name="spellcheck">
  <lst name="suggestions">
    <lst name="julius">
      <int name="numFound">2</int> ...
      <arr name="suggestion">
        <str>justus</str>
        <str>julian</str>
      </arr>
    </lst>
    <lst name="collation">
      <str name="collationQuery">justus</str>
      <int name="hits">2</int>
      ...
    </lst>
    ...
  </lst>
</response>
```

为使搜索结果简短，该例中仅返回 title 字段。

找到 4 个匹配 julius 的文档。

发现 2 个搜索建议，justus 与 julian。

建议词 justus 仅出现在 2 个文档中。

这里，Solr 为查询 Julius 找到相关结果并返回拼写建议。注意，Julius 返回的结果数超过了建议词返回的结果数。Julius 匹配了 4 个文档，而建议词 justus 匹配了 2 个文档。在这种情况下，不显示“你是不是要找 justus?”可能是个好主意，因为 justus 出现的频次比 Julius 低。关键的一点是，搜索引擎需要决定什么时候应该向用户显示“你是不是要找……?”。在大多情况下，如果查询词与建议词的搜索结果情况差不多，可能就不需要显示这个提示信息了。至此，我们已经了解 Solr 的拼写检查用法，下面深入了解一下它的工作原理。



### 10.1.3 拼写检查搜索组件

在代码清单 10.1 至 10.3 中的查询请求中，注意到一点，这些查询都是通过使用第 4 章介绍的 `/select` 请求处理器进行提交。同时还须注意，虽然我们已明确指定拼写检查支持，但查询中没有出现用于拼写检查的参数。这是因为拼写检查组件已经集成在 `solrconfig.xml` 定义的默认 `/select` 请求处理器中，具体参见代码清单 10.4。搜索请求处理器及组件的相关内容，请参见 7.1 节。

代码清单 10.4 在 `/select` 请求处理器中集成拼写检查功能

```
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>
    <str name="df">text</str>
    <str name="spellcheck">on</str>
    <str name="spellcheck.extendedResults">>false</str>
    <str name="spellcheck.count">5</str>
    <str name="spellcheck.alternativeTermCount">2</str>
    <str name="spellcheck.maxResultsForSuggest">5</str>
    <str name="spellcheck.collate">>true</str>
    <str name="spellcheck.collateExtendedResults">>true</str>
    <str name="spellcheck.maxCollationTries">5</str>
    <str name="spellcheck.maxCollations">3</str>
  </lst>
  <arr name="last-components">
    <str>spellcheck</str>
  </arr>
</requestHandler>
```

在 `solrconfig.xml` 中配置 `/select` 请求处理器。

启用拼写检查，使用默认配置。

微调拼写检查行为，参见表 10.1。

最后执行拼写检查组件。

简言之，之所以将拼写检查集成到 `/select` 请求处理器，因为这是所有查询默认启用的核心功能。此外，它为搜索应用减轻了在每个查询中传递所有拼写检查参数的负担。如果某查询不希望启用拼写检查，设置 `spellcheck=false` 即可。由于仍要在查询处理中执行默认搜索组件，如 `query`、`facet` 和 `debug`，所以在最后执行拼写检查组件有一定道理。如果设置 `spellcheck.collate=true` 启动校对，拼写检查组件则必须列为后处理模块，因为生成校对查询要求查询组件已经执行。一般情况下，建议将拼写检查组件配置为请求处理器的后处理模块。

表 10.1 是拼写检查的可用参数一览表，将拼写检查集成到请求处理器时可指定这些参数。

表 10.1 拼写检查参数介绍

拼写检查参数	说明
spellcheck	对查询请求启用 (true) 或禁用 (false) 拼写检查
spellcheck.q	为查询指定拼写检查。如果未提供该参数, Solr 对 q 参数应用拼写检查
spellcheck.build	如果为 true, 指示 Solr 建立拼写检查字典 (若不存在的话)。由于字典基于主索引, 所以 DirectSolrSpellChecker 不需要此选项
spellcheck.reload	如果为 true, 导致 Solr 重新加载拼写检查器。这对 DirectSolrSpellChecker 无效, 它始终保持最新
spellcheck.count	指定要提供的拼写建议最大数 (整数值)
spellcheck.dictionary	设置字典名称, 用于生成建议, 例如 default。在一个请求中可以激活多个拼写检查器, 每个字典都包含此参数。具体示例参见代码清单 10.8
spellcheck.onlyMorePopular	如果为 true, 对建议查询进行限制, 要求建议查询比原始查询匹配的文档要多
spellcheck.maxResultsForSuggest	在 Solr 尝试生成建议之前, 对匹配结果的数量设置阈值 (整数)。这样可以有效控制匹配出足够多文档的查询建议。如果设置此参数为 10, 初始查询匹配了 12 个文档, 则拼写建议被禁用
spellcheck.accuracy	介于 0 到 1 之间的浮点数, 以确定查询建议是否有效。值越大表示精确度越高, 但查询建议越少
spellcheck.extendedResults	获取查询建议词项的其他信息, 例如, 词项文档频次
spellcheck.collate	如果为 true, Solr 根据建议的拼写纠错生成一个替代查询。如果用户点击“你是不是要找……?”链接, 搜索引擎执行校对查询。校对查询一定会返回搜索结果。这就是说, Solr 返回结果之前, 在后台得执行校对查询
spellcheck.maxCollations	限制 Solr 生成的校对查询数 (整数值)
spellcheck.maxCollationTries	校对尝试最大数 (整数值)。在所有情况下都不提供建议, 取值越小效果越好

在代码清单 10.4 中, spellcheck 搜索组件作为最后一个组件, 集成在 /select 请求处理器中。4.2.4 节曾介绍过, 最后一个组件在内置的核心组件 (如 query、facet 等) 之后执行, 参见图 4.5。代码清单 10.5 是 solrconfig.xml 中 spellcheck 搜索组件的配置信息。

代码清单 10.5 在 solrconfig.xml 文件中配置拼写检查搜索组件

```

从 suggest 字段中创建一个词项字典。
<searchComponent name="spellcheck" class="solr.SpellCheckComponent"
  <str name="queryAnalyzerFieldType">text_suggest</str>
  <lst name="spellchecker">
    <str name="name">default</str>
    <str name="field">suggest</str>
    <str name="classname">solr.DirectSolrSpellChecker</str>
    <str name="distanceMeasure">internal</str>
    <float name="accuracy">0.5</float>
    ...
  </lst>
  <lst name="spellchecker">
    <str name="name">wordbreak</str>
    <str name="classname">solr.WordBreakSolrSpellChecker</str>
    <str name="field">suggest</str>
    <str name="combineWords">true</str>
    <str name="breakWords">true</str>
    <int name="maxChanges">10</int>
    <int name="minBreakLength">5</int>
  </lst>
  ...
</searchComponent>

```

在 schema.xml 中定义字段类型，用来分析查询词项。

使用 Levenshtein 字符串距离找到相似词项。

建议的精准度，取值介于 0 与 1 之间。

另外的拼写检查，名为“wordbreak”。

使用主索引生成建议。

代码清单 10.5 对 spellcheck 组件进行配置，根据主索引的 suggest 字段上的词项提供查询建议。接下来，我们从 DirectSolrSpellChecker 组件逐一解释该配置的几个部分。

### DirectSolrSpellChecker

Solr 4 的默认拼写检查组件是 DirectSolrSpellChecker 组件。该组件得名于它根据主索引直接提供建议。而在 Solr 的早期版本中，需要基于主索引构建一个单独的拼写检查索引。基于主索引提供建议的做法可以避免，在主索引改变后需要重新构建所有二级索引的情况，因此该方法优于维护二级索引。

DirectSolrSpellChecker 组件可以调整多个参数，其中最重要的三个参数分别是 field、distanceMeasure 和 accuracy。field 参数指定索引中用于提供建议的字段。在本例中，在 schema.xml 文件中定义为 suggest 字段，如下所示：

```
<field name="suggest" type="text_suggest" indexed="true" stored="false"/>
```

suggest 字段使用 <copyField> 中的 title 字段进行填充：

```
<copyField source="title" dest="suggest"/>
```

通过使用复制字段，可将不同的文本分析策略应用到维基百科文章的 title 字段上。另外，还可以从其他字段（如 article 文本字段）添加词项，将更多的词项

添加到拼写检查字典中。关于如何将词项从 article 文本字段添加到 suggest 字段, 请读者自行探索。

distanceMeasure 参数告诉 Solr 如何确定查询词的建议。拼写检查如何在字典中识别出词项, 这超出了本书的介绍范围, 但这个过程背后的原理是很简单的。本书第 3 章和第 7 章讨论模糊查询时提到, 拼写检查组件使用一些方法来计算查询词和字典中的每个词之间的字符串距离(编辑距离)。计算两个词项之间距离的一种方法是, 考虑一个词项要做出多少变化才能变成另一个词项。例如, atmosphear 和 atmosphere 之间的编辑距离为 2, 一个变化是去掉字母 a, 另一个变化是在单词结尾加字母 e。计算两个词之间距离的一个著名算法是 Levenshtein 距离<sup>2</sup>, 将 distanceMeasure 参数设置为 internal 就可以使用 Levenshtein 距离算法。

accuracy 参数是一个介于 0 和 1 之间的浮点值, 它确定了查询建议需要的准确程度。数字越大, 准确度越高, 但会导致匹配结果减少, 这种情况下可能没有可用的查询建议。如果 accuracy 设定值太低, Solr 会产生很多查询建议, 但无法保证这些查询建议对用户而言是有意义的。

接下来介绍 queryAnalyzerFieldType 的配置, 它对用于分析查询词和生成建议的字段类型进行定义。

### 用于拼写检查的文本分析

考虑一下在拼写检查字典的词项上应执行的文本分析类型。在大多数情况下, 我们都希望文本分析力度尽可能小, 尤其要避免过滤器对词项进行大幅度修改, 例如, 词干提取或语音分析。代码清单 10.6 使用了 schema.xml 定义的 text\_suggest 字段类型, 显示了 text\_suggest 字段类型的相关内容。

代码清单 10.6 用于拼写检查的词项分析的 text\_suggest 字段类型

```
<fieldType name="text_suggest" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
      words="stopwords.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ASCIIIFoldingFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
  </analyzer>
</fieldType>
```

如果你阅读过第 6 章, 理解这段代码没什么问题。这个字段类型在空白和标点处分割文本、保留 URL 和 email 地址 (UAX29URLEmailTokenizer)、移除

2 Levenshtein 距离, [http://en.wikipedia.org/wiki/Levenshtein\\_distance](http://en.wikipedia.org/wiki/Levenshtein_distance)。

停用词 (StopFilter)、转换为小写 (LowerCaseFilter)、将拉丁字符转换成对应的 ASCII 值 (ASCIIFoldingFilter), 并且移除了英语中的所有格形式 (EnglishPossessiveFilter)。从拼写检查角度看, 所有这些操作都是安全转换。此外, 在索引和查询过程中也采用了相同的文本分析。在代码清单 10.1 的示例查询中, 查询 Julius 变成了查询 julius。对于输入包含重音字符的查询词 (如 jalapeño) 的用户, ASCIIFoldingFilter 提供的建议是很有用的, 因为用户在构建查询时输入重音形式非常少见。

在代码清单 10.5 中, DirectSolrSpellChecker 组件的配置使用 suggest 字段来构建自己的字典, 可以猜测出 suggest 字段也要使用 text\_suggest 字段类型。一般情况下, 拼写检查的查询词分析需要与构建拼写检查字典采用相同的分析策略, 这样能够确保拼写检查组件在字典和查询词之间使用兼容的分析模式。

### 其他拼写检查实现方式

Solr 拼写检查的强大特征之一是可以将多个拼写检查实现方法结合起来创建复合拼写检查方法。例如, 在代码清单 10.5 中, WordBreakSolrSpellchecker 组件配置为一个额外的拼写检查方法。WordBreakSolrSpellchecker 组件通过将查询词分为多个部分或组合查询词来找到查询建议。以一个拼写错误的查询为例, 如 northatlantic curent, 用户原本打算查找有关 North Atlantic Current 的话题。在代码清单 10.7 中一直使用的默认拼写检查组件会顺利地对 curent 生成一个有效的查询建议, 但没有为 northatlantic 提供查询建议。

#### 代码清单 10.7 默认拼写检查组件对处理应该被分开的词存在问题

##### 查询请求

```
http://localhost:8983/solr/solrpedia/select?
q=northatlantic curent&df=suggest&wt=xml
```

##### 搜索结果

```
<response>
<lst name="responseHeader">...</lst>
<result name="response" numFound="0" start="0"></result>
<lst name="spellcheck">
  <lst name="suggestions">
    <lst name="curent">
      ...
      <arr name="suggestion">
        <str>curent</str>
        <str>cent</str>
      </arr>
    </lst>
```

拼写检查器为“curent”找到搜索建议, 没有为“northatlantic”提供任何搜索建议。

```

<lst name="collation">
  <str name="collationQuery">northatlantic current</str>
  <int name="hits">1</int>
  <lst name="misspellingsAndCorrections">
    <str name="curent">current</str>
  </lst>
</lst>
...
</lst>
</lst>
</response>

```

用户没有在 northatlantic 之间输入空格，但我们需要 Solr 能识别出 northatlantic 实际上是两个词。一般情况下，在查询条件之间乱放空格是一种常见的用户输入错误，因此应该在拼写检查解决方案中纠正这个错误。WordBreakSolrSpellchecker 组件能够把查询词分成多个部分，然后在索引中查找分开后的词项，从而提供正确的查询建议。代码清单 10.8 是 WordBreakSolrSpellchecker 组件的一个使用举例。

代码清单 10.8 同时调用 wordbreak 拼写检查组件和默认拼写检查组件的查询请求

#### 查询请求

http://localhost:8983/solr/solrpedia/select?

q=northatlantic curent&  
wt=xml&df=suggest&q.op=AND&  
spellcheck.dictionary=wordbreak&  
spellcheck.dictionary=default

激活 wordbreak  
拼写检查器。

激活默认拼写检查  
器。

#### 搜索结果

```

<response>
...
<result name="response" numFound="0" start="0"></result>
<lst name="spellcheck">
...
  <lst name="collation">
    <str name="collationQuery">(north atlantic)
      current</str>
    ...
  </lst>
</lst>
</response>

```

在两种拼写检查  
器共同作用下找  
到最佳建议。

代码清单 10.8 展示了在一个不佳的查询中，如何同时使用默认拼写检查组件和 wordbreak 拼写检查组件来提供一个准确的查询建议。在这种情况下，搜索引擎在后台执行 collationQuery，但不给用户提示，向用户显示搜索 north atlantic current 的搜索结果。这里的关键点在于，如何将多个拼写检查方法整合成一个统一的解决方案，Solr 在这一方面提供了很大的灵活性支持。

至此，我们已经熟悉了 Solr 的拼写检查组件，接下来学习自动建议，它使用了许多与拼写检查相同的技术。

## 10.2 自动建议查询词

拼写检查是个不错的功能，不过还可以根据用户的输入给出建议查询词，从最开始就避免拼写错误。在移动环境中“胖手指失误”（fat-finger errors）很常见，自动建议就能发挥重要作用。本节重点介绍 Solr 4 内置的建议组件（Suggester）。

先从概念上搞清楚自动建议是怎么回事，这有助于理解 Solr 的建议组件的工作原理。自动建议的用户界面很简单，就是一个根据用户输入显示的建议列表。在图 10.3 中，每增加一个字符都会改变建议列表。

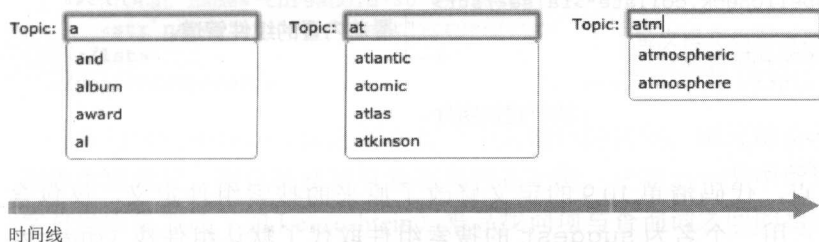


图 10.3 基于 solrpedia 索引随输入变化的建议查询词

一般情况下，自动建议功能需要满足两个要求：

- 一定要快。一个笨重的输入方案不能跟上用户的输入节奏，没有比这更恼火的事情了。建议组件必须保证随着用户输入的每个字符更新建议，这种响应达到毫秒级。
- 根据词频顺序返回排名建议。如果建议的是索引中只有少数文档出现的罕见词，特别是当用户只输入少量字符时，这种建议没有什么意义。

搞清楚自动建议是什么之后，下面设计一个自定义的请求处理器来支持自动建议，让我们看一下 Solr 中如何使用自动建议。

### 10.2.1 自动建议请求处理器

本节介绍如何使用 Solr 内置的建议组件，为 solrpedia 实现自动建议。首先，需要定义特定的用于生成建议的请求处理器。在上一节中，我们将拼写检查集成到默认的 /select 请求处理器。事实上，这种设计方法不是处理自动建议的最佳途径。一方面，不需要向 /select 请求处理器添加拼写检查支持；另一方面，不需要任何内置的组件，例如，query、facets 和 highlighting。与面向对象编程类似，



Solr 专门设计了自定义的请求处理器，用来封装简单界面背后的复杂行为。代码清单 10.9 定义了一个支持自动建议的自定义请求处理器。

代码清单 10.9 在 solrconfig.xml 文件中定义 /suggest 请求处理器

```
<requestHandler name="/suggest"
  class="org.apache.solr.handler.component.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">none</str>
    <str name="wt">json</str>
    <str name="indent">>false</str>
    <str name="spellcheck">>true</str>
    <str name="spellcheck.dictionary">suggestDictionary</str>
    <str name="spellcheck.onlyMorePopular">>true</str>
    <str name="spellcheck.count">5</str>
    <str name="spellcheck.collate">>false</str>
  </lst>
  <arr name="components">
    <str>suggest</str>
  </arr>
</requestHandler>
```

自定义请求处理器仅执行自动建议。

识别用于搜索建议的字典。

覆盖内置的组件管道。

执行建议组件。

应该注意一点，代码清单 10.9 的定义修改了原来的搜索组件定义，仅包含建议搜索组件。它用一个名为 suggest 的搜索组件取代了默认组件栈（query、facet、debug 等）。请记住一点，自动建议的关键要求是执行速度快，因此我们不希望 /suggest 处理器在执行其他搜索组件上消耗太多资源。

为了收到查询建议，搜索应用必须发送请求给 /suggest 处理器。代码清单 10.10 给出了 /suggest 请求处理器的使用示例。

代码清单 10.10 /suggest 请求处理器实际运用示例

#### 查询请求

http://localhost:8983/solr/solrpedia/suggest?q=atm

#### 搜索结果

```
{
  "responseHeader": {...},
  "spellcheck": {
    "suggestions": [
      "atm", { "numFound": 2,
        ...
        "suggestion": ["atmosphere", "atmospheric"]
      }
    ]
  }
}
```

发送 atm 前缀到 /suggest 请求处理器。

solrpedia 索引中 suggest 字段的搜索建议。

现在，我们来看如何通过 /suggest 请求处理器来使用建议搜索组件。

## 10.2.2 自动建议搜索组件

当 /suggest 处理器收到一个请求时，它会调用建议搜索组件来生成建议。代码清单 10.11 是 solrconfig.xml 文件中建议搜索组件的配置。

代码清单 10.11 solrconfig.xml 文件中建议搜索组件的定义

```
<searchComponent class="solr.SpellCheckComponent" name="suggest">
  <lst name="spellchecker">
    <str name="name">suggestDictionary</str>
    <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
    <str name="lookupImpl">org.apache.solr.spelling.suggest.fst
      .FSTLookupFactory</str>
    <str name="field">suggest</str>
    <float name="threshold">0.</float>
    <str name="buildOnCommit">true</str>
  </lst>
</searchComponent>
```

对搜索建议词典进行命名。

构建在拼写检查框架之上。

查找的实现类。

基于 suggest 字段。

每次提交后需重建查找的数据结构。

拼写检查和自动建议之间存在一个关键性的区别。拼写检查从一个完整的查询词中生成建议，而自动建议只具有预修正功能。正如上一节中所述，拼写检查使用字符串距离方法（如 Levenshtein）来寻找词项与查询词之间的相似性。自动建议并没有一个完整的词来生成建议，因此字符串距离并没有太大的意义。

实现自动建议的一个原始方法是，当用户输入时使用通配符搜索。然而，对自动建议而言，通配符搜索速度太慢，需要另寻其他方法。Solr 内置的建议组件使用前缀树的数据结构，它支持使用前缀进行快速查找，这正是预输入建议所需要的。

Solr 的查找结构对通过前缀查找词项是有效的，当用户输入 at 时，Solr 能快速地找到以 at 开头的很多词。在大型索引中，一个前缀可以匹配出许多词项，这时我们不希望低频词和罕见词对建议列表造成干扰，因此需要一种方法，按照热门程度对查询建议词进行排序。从大多数用户角度考虑，如果一个词项仅在数百万个文档中的几个文档中出现过，那么它可能不是一个好的查询建议。

代码清单 10.11 中使用了 org.apache.solr.spelling.suggest.fst.FSTLookup 类，它的数据结构基于有限状态自动机（Finite State Automaton, FSA），无论前缀长短，能够进行快速、即时的查找。FSTLookupFactory 方法构建速度相对较慢，但它的内存占用量很小，对于大型字典而言是一个不错的选择。

当索引新的文档时，建议字典必须重新构建，从而添加新词。在代码清单 10.11 的配置中，每次提交后需要重新构建字典（buildOnCommit=true）。该字典保存在内存中，构建速度非常快，因此它通常不会带来执行性能问题。

### 10.3 文档字段值建议

建议组件能够根据用户的输入给出查询词建议，对于词组或短字段（如标题）则效果并不好。本节介绍如何从文档中给出建议字段值的另一种方法。

在 solrpedia 例子中，根据用户的输入可以给出主题标题建议。假设用户输入 river，使用上一节介绍的建议组件将提示 rivers、riverdale 和 riverside。本节介绍的方法能显示一个包含词 river 的主题列表，如图 10.4 所示。

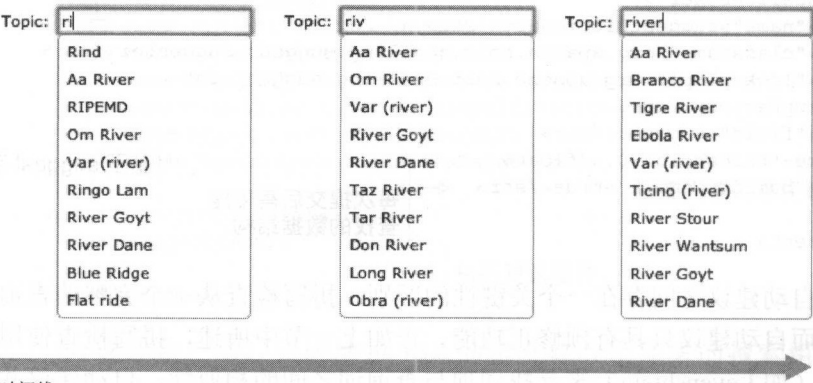


图 10.4 根据用户的输入给出相应的建议主题

这两种方法对搜索应用都有效，但就 solrpedia 例子而言，主题建议对特定的搜索应用更为合适。构建搜索应用时，需要考虑的是哪种方法对用户而言更有意义。接下来介绍图 10.4 中搜索建议的实现方法。

#### 10.3.1 使用 n-grams 生成建议

正如图 10.4 所示，主题建议组件匹配的主题并不一定是以字母 riv 开始的。只要 riv 前缀出现在标题中，就能生成建议。实现这种效果的一个方法是，使用通配符查询 suggest 字段，例如，riv\*。但这种方法存在两个问题。通配符查询的执行速度很慢（前面提到过），尤其是在包含众多词项的字段上进行查询。另外，使用通配符也无法控制返回文档的排序。

更好的一种方法是，在文本分析过程中对每个词项创建边缘（edge）n-grams。n-gram 是由一个单词或字符串生成的连续字符序列，其中 n 表示序列的长度。例如，1-gram 是一个单字组，表示 n-gram 的长度是 1。同样道理，双字组是包含两个连续字符的序列。单词 river 的双字组包含 ri、iv、ve 和 er。边缘 n-grams 是 n-gram 的一种特殊类型，它由单词的一侧生成，对英文单词来说通常是左

侧。单词 river 的边缘 n-grams 包括 r、ri、riv、rive 和 river。Solr 使用 EdgeNGramFilter 在文本分析过程中为单词生成边缘 n-grams。

为了生成建议，n-grams 与事先计算好的通配符搜索类似，这样做有效避免了上述性能问题。查询时，在 suggest 字段搜索前缀 riv，可以得到精确匹配 riv 的 3-gram 中包含词 river 的所有主题。如果使用 EdgeNGramFilter 来分析主题名称，每个词所有可能的前缀都会被索引。由于为每个索引的词都创建了许多 n-grams，使用 n-grams 会大大增加索引的大小。示例中只对主题字段生成 n-grams，这样做一般只包含两至三个词，因此对索引大小的影响是可控的。

如代码清单 10.12 所示，在 schema.xml 文件中定义一个名为 text\_suggest\_ngram 的字段类型，在文本分析时创建 n-grams。

代码清单 10.12 在文本分析期间用于创建一般 n-grams 的字段类型

```
<fieldType name="text_suggest_ngram"
  class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
      words="stopwords.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ASCIIFoldingFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.EdgeNGramFilterFactory"
      maxGramSize="10" minGramSize="2"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
      words="stopwords.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ASCIIFoldingFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
  </analyzer>
</fieldType>
```

仅在索引创建阶段的文本分析中，使用分词过滤器生成 n-grams。

text\_suggest\_ngram 字段类型的一个重要方面是，它对索引和查询使用不同的但又兼容的分析策略。该字段类型的使用是通过用户输入的前缀产生建议，所以只需要在索引期间生成 n-grams 即可。此外，为了避免将 n-grams 注入到 Solr 建议组件使用的 suggest 字段中，定义一个新字段来使用 text\_suggest\_ngram 字段类型，通过 <copyField> 进行字段填充。

```
<field name="suggest_ngram" type="text_suggest_ngram"
  indexed="true" stored="false"/>
<copyField source="title" dest="suggest_ngram"/>
```

很明显，我们不希望使用 `suggest_ngram` 字段作为词项层次的建议来源。因为这可能会生成不完整的建议词（如 `rive`）。接下来，让我们来看如何使用 `suggest_ngram` 字段生成图 10.4 中所示的结果。

### 10.3.2 n-gram-driven 请求处理器

我们创建一个搜索请求处理器，使用 `n-grams` 来生成建议标题。考虑如何配置请求处理器来使用 `n-grams` 时，需要注意一点，某些 `n-grams` 可能是完整的词。在大多数情况下，我们希望将精确匹配结果排在部分匹配结果之前。如图 10.5 所示，如果用户输入 `long`，显示结果中精确匹配的词 `Long Valley` 和 `Long River` 将位于部分匹配的词 `Longo` 和 `Longitude` 之前。

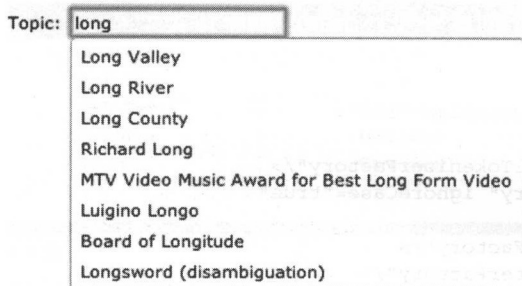


图 10.5 对建议排序，精确匹配结果比部分匹配结果的权重高

在代码清单 10.13 的搜索请求处理定义中，使用 Solr 的 `eDisMax` 查询解析器来实现预期结果，这一做法在 7.5 节讨论过。

#### 代码清单 10.13 使用 `eDisMax` 查询解析器提升精确匹配结果的权重

```
<requestHandler name="/suggest_topic"
  class="org.apache.solr.handler.component.SearchHandler">
  <lst name="defaults">
    <str name="wt">json</str>
    <str name="defType">edismax</str>
    <str name="rows">10</str>
    <str name="fl">title</str>
    <str name="qf">suggest^10 suggest_ngram</str>
  </lst>
</requestHandler>
```

使用最大编辑距离查询解析器。

返回 title 字段，显示搜索建议。

调整精确匹配的权重高于部分 n-grams 匹配。

在代码清单 10.13 中，将 `suggest` 字段上精确匹配的权重设置为 `ngram` 字段上其他匹配的权重的 10 倍。关键点在于，此处使用 Solr 的内置文档评分框架为用户输入的查询找到最佳建议。同时还应清楚一点，由于对多个字段进行匹配，所以

可以通过定义额外的字段来使用任意数量的文本分析策略，并对这些字段设置不同的权重。例如，根据语音匹配生成建议，假设用户输入 `highbred`，可以建议 `hybrid`。这种方法和内置的基于索引词项频次相比较，应该选用哪一种，视具体情况需求而定。

在结束本章之前，介绍另一种基于用户过去活动来生成查询建议的方法。

## 10.4 基于用户活动提供查询建议

本章已经介绍的查询建议方法都存在一个缺点，即它们没有考虑用户过去输入的查询。10.2 节的内置建议组件擅长处理查询词建议，10.3 节的基于 `n-gram` 建议组件擅长从文档中使用部分词匹配来生成建议字段。然而，这两种方法提供的查询建议没有考虑到基于搜索引擎用户行为的查询词的热门程度。

基于用户过去查询行为的查询建议是集体智慧的一种表现形式。在使用谷歌即时搜索时<sup>3</sup>，你可能已经看到过此种类型的查询建议。其基本思想是，当用户输入时，执行最可能的查询，该查询选自于最近热门查询数据库。假设用户输入 `justin b`，基于集体智慧的建议引擎可能会返回关于 Justin Bieber 的搜索结果，这是因为他是网络上流行的名人。特别是，当用户正在搜索热门话题时，这种查询建议可以减少用户输入，并且看起来很智能。

Solr 的实现方式与 10.3 节的 `n-gram` 配置类似。但是，需要对一个 Solr 次索引进行查询，该索引包含来自查询日志的用户查询，而不是去匹配主索引的字段。你需要开发一个查询日志分析工具，计算每个查询的热门程度得分。为了更新热门程度得分，并纳入新的查询分析必须持续进行。通过查询 Solr 次索引来生成建议时，需要将查询的热门程度纳入到得分中，或者按热门程度按降序排列。

### schema 设计

查询日志挖掘留给读者自行探索。为简单起见，假设有表 10.2 的数据，它们是从 `solrpedia` 查询日志中挖掘而来。

表 10.2 从查询日志挖掘的查询热门程度数据示例

查询	最后执行日期	频次（过去 30 天）
Prince William	August 10, 2013	20 000
Fort William	July 15, 2013	20 005
William and Mary	July 22, 2013	19 995

3 谷歌即时搜索有关内容请参见 <http://www.google.com/insidesearch/features/instant/about.html>。



我们在一个名为 `solrpedia_instant` 的二级 Solr 内核中存储查询及其热门程度数据。代码清单 10.14 显示了这个二级索引的 `schema.xml` 文件中的主要字段。

代码清单 10.14 `solrpedia_instant` 内核中用于建议热门查询的字段

```
<field name="id" .../>
<field name="query" type="string" indexed="false" stored="true"/>
<field name="query_ngram" type="text_suggest_ngram" indexed="true" stored="false"/>
<field name="popularity" type="tfloat" indexed="true" stored="true"/>
<field name="last_executed_on" type="tdate" indexed="true" stored="true"/>
<field name="_version_" .../>
```

从查询日志中挖掘出原始查询文本。

包含查询文本 n-grams 的字段 (参见 10.3 节)。

查询日志分析后得到热门程度得分。

在查询日志中找到最近一天的查询。

请注意, `query_ngram` 字段使用 10.3 节中介绍的 `text_suggest_ngram` 字段类型。根据代码清单 10.14 的 `schema` 设计, 对表 10.2 的数据以 JSON 方式进行索引。

```
[
  {
    "id" : "1",
    "query" : "Prince William",
    "last_executed_on" : "2013-08-10T00:00:00Z/DAY",
    "popularity" : 20000
  },
  {
    "id" : "2",
    "query" : "Fort William",
    "last_executed_on" : "2013-07-15T00:00:00Z/DAY",
    "popularity" : 20005
  },
  {
    "id" : "3",
    "query" : "William and Mary",
    "last_executed_on" : "2013-07-22T00:00:00Z/DAY",
    "popularity" : 19995
  }
]
```

执行以下命令, 将这些文档导入到 Solr 中:

```
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/solrpedia_instant/update
  -Dtype=application/json -jar post.jar ch10/documents/
  solrpedia_instant.json
```

至此, 我们已经设计了基础的二级索引, 接下来学习如何在主索引上找到最热门的查询并执行该查询。



### 寻找最热门的查询

在激活即时建议之前，用户需要输入多少个字符？与查询词的预输入建议有所不同，这里是根据用户的过去行为来预测用户想要查什么。因此，在预测用户的意图之前，可能需要用户输入更多的字符。例如，在 solrpedia 例子中，用户输入 6 个字符之后可能才会启动即时建议。在输入 6 个字符前，只能依赖前面章节所讲的预输入建议。

根据表 10.2 的数据，假设用户输入 willia，建议引擎将匹配热门查询 Fort William，然后即时显示结果。代码清单 10.5 举例说明了如何使用 solrpedia\_instant 内核，根据用户的输入寻找建议查询。

代码清单 10.15 基于热门程度寻找建议查询的查询示例

```
http://localhost:8983/solr/solrpedia_instant/select?
q=query_ngram:willia&
sort=popularity desc&
rows=1&
fl=query&
wt=json
```

从查询日志中  
返回查询文本。

仅返回一个  
搜索建议。

按照热门程  
度降序排  
列。

查询包含  
n-grams  
的字段。

```
{
  "response": {
    "numFound":3,
    "start":0,
    "docs":[
      { "query":"Fort William" }
    ], ...
  }
}
```

根据表 10.2 的数据，  
查询建议是“Fort  
William”。

在二级索引上查询 willia 返回了 Fort William，由于它的热门程度得分是 20 005（参见表 10.2），从而成为查询建议。除了热门程度，如果想要根据每个查询的热门程度得分的新旧程度，对结果排序分组，又该怎么做？如何考虑查询热门程度的新旧程度呢？

### 提升最近最热门查询的权重

考虑查询的热门程度的同时，热门程度得分的新旧程度也很重要。因此，需要提升最近最热门查询的权重，同时降低那些以前流行现在不再流行的热门查询的权重。从表 10.2 可以看出，查询 Fort William 比查询 Prince William 更热门，但相比之下，查询 Fort William 不够新。因此，如果要考虑查询热门程度的新旧程度，当用户输入 willia 时，Prince William 会是一个更好的建议，这是因为它比 Fort William 的查询日期更新一些。

为什么不将新旧程度纳入到热门程度得分的计算中呢？新旧程度每天都在变

化,如果每天都要在二级索引上重新索引所有文档,这对繁忙的搜索引擎而言影响是很大的。更好的一种方法是,按照每个文档最后执行的日期进行索引,使用 Solr 函数查询动态计算新旧程度的权重。

假设在二级索引中保存过去 30 天的查询。当每天处理查询日志时,并不是二级索引中的每个查询都会在日志中显示。为了考虑查询热门程度得分的新旧程度且不要每次重新索引每个查询,这需要在索引中每个文档的日期字段上保存最新一次查询的日期。这个字段的值是日志中给定查询的最新一次执行日期。比方说,Prince William 查询在 2013 年 8 月 9 日产生了 19 980 次查询,当处理 2013 年 8 月 10 日的日志时,发现多出 20 多个查询。这时你可能会想用下面的数据更新一下二级索引:

```
{
  "id" : "1",
  "query" : "prince william",
  "last_executed_on" : "2013-08-10T00:00:00Z/DAY",
  "popularity" : 20000
}
```

通过更新 last\_executed\_on 字段,显示给定查询词在日志中最新一次的执行日期,热门程度的得分是超过 30 天的查询合计数(20 000 = 之前的 19 980 + 现在的 20)。如代码清单 10.16 所示,使用 Solr 的 boost 函数查询,按照热门程度和新旧程度来提升文档的权重。

代码清单 10.16 使用 Solr 的 boost 函数按热门程度和新旧程度来提升文档权重

```
http://localhost:8983/solr/solrpedia_instant/select?
q={!boost b=$recency v=$qq}&
sort=score desc&
rows=1&
wt=json&
qq=query_ngram:willia&
recency=product(recip(ms(NOW/HOUR,last_executed_on),
1.27E-10,0.08,0.05),popularity)
```

根据得分降序排列。

使用局部参数调整权重。

局部 qq 参数搜索 query\_ngram 字段。

局部 recency 调整 popularity 与 age 权重。

该查询的结果是,尽管 Fort William 的热门程度得分更高,但 Prince William 作为最佳建议被返回。这是因为 boost 函数使用一个基于时间的衰减函数来降低较旧的热门程度得分。表 10.3 对代码清单 10.6 中查询请求的各部分分别解释,这样能更好地理解其背后的工作原理。

表 10.3 按新旧程度和热门程度提升文档权重

查询参数	解释
<code>q={!boost b=\$recency v=\$qq}</code>	权重查询解析器通过 <code>b</code> 参数指定的函数查询，提升与 <code>v</code> 参数指定查询相匹配的文档权重。Solr 本地参数可简化该查询的语法，参见 7.2.2 节
<code>sort=score desc</code>	按照得分降序排列，这是默认的排序因素。这里为了清楚说明，根据调整的分进行排序
<code>qq=query_ngram:willia</code>	当用户输入 <code>willia</code> ，使用局部参数语法，搜索 <code>query_ngram</code> 字段
<code>recency=product(   recip(ms(NOW/HOUR,     last_executed_on),     1.27E-10,0.08,0.05),   popularity)</code>	计算热门程度与时间的权重积。时间权重使用 Solr 的 <code>recip</code> 函数： $\text{recip}(x,m,a,b) = a / (m \times x + b)$ ，其中 $x$ 是文档的新旧程度。 $m$ 、 $a$ 与 $b$ 参数用于计算时间惩罚。文档的新旧程度是当前时间与 <code>last_executed_on</code> 字段值之间间隔的毫秒数

图 10.6 显示了 `boost` 函数如何根据新旧程度来降低查询的热门程度得分。通过改变参数  $m$ 、 $a$  和  $b$  的值来影响 `boost` 函数的作用力度。本节选择  $m=1.27\text{E}-10$ 、 $a=0.08$  和  $b=0.05$  来快速降低较旧的热门程度得分。

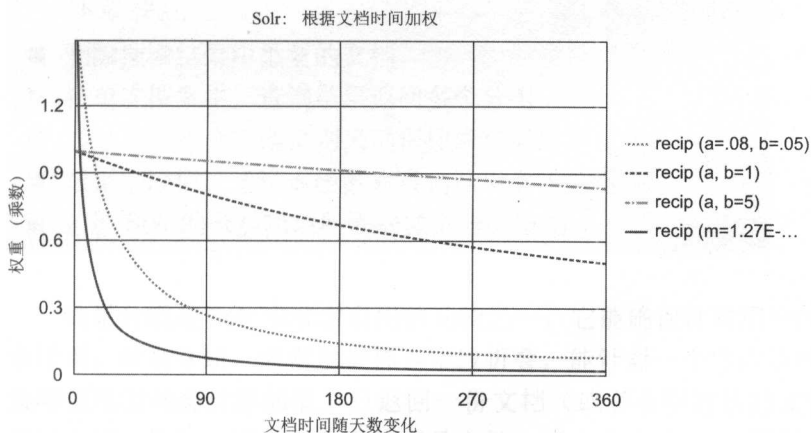


图 10.6 Solr 的 `recip` 函数根据文档的新旧程度提升或降低查询的热门程度得分，任何低于 1 的值都会降低其权重

查询二级索引之后，搜索应用可以在后台将建议查询发送给主索引。这会让用户感觉到，搜索引擎正在根据他们的输入预测他们想要搜索的内容。这里需要提醒 Solr 开发者，这个功能会大幅提升主索引的查询量，这是因为在用户构造好完整的查询之前，搜索应用正在后台执行查询。与任何用户体验相关的功能一样，这需要监控实际的用户行为，以此确定搜索引擎的这个功能是会提升还是降低用户的工作效率。

## 10.5 本章小结

本章通过查询建议来提高搜索应用的可用性。具体来说，本章介绍了如何实现“你是不是要找……”形式的查询建议来处理拼写错误的查询词。然后，介绍了使用 Solr 内置的建议组件来根据用户的输入返回查询建议。这两种方法是互补的，通常一起使用，构成一个完善的查询建议解决方案。

接下来介绍了另一种方法，通过生成建议主题来替代原始查询词。在索引过程中，使用 Solr 的 EdgeNGramFilter 生成 n-grams，从而可以有效地使用前缀来查询那些建议文档。本章结束之前，还简要讨论了如何使用 Solr 实现即时搜索建议功能。

下一章将介绍搜索结果分组。

# 结果分组/字段折叠

## 本章要点

- 删除搜索结果中重复的文档
- 在单次搜索中，查询结果返回多个分组
- 通过显示多个文档类别来确保搜索结果的多样化
- 根据字段值、查询或函数对查询结果进行分组
- 扩展 Solr 的分组功能所需的数据分区策略

结果分组是 Solr 中非常有用的功能之一，它能确保针对用户的查询返回最佳搜索结果。结果分组，通常也被称为字段折叠，能针对一个字段里的唯一值（当前的实际值或者动态计算的值）只返回一份文档（或者有限数量的文档）。如果有多份相似文档，例如，同一家公司的产品文档、同一家连锁餐厅的地址文档或者某公司的多个办公地点信息的文档，这个功能就能派上用场。但是，也不要希望整页的搜索结果只展示一种产品、一家餐厅或者一家公司。

你在使用自己偏爱的网络搜索引擎时，可能已经见到过这个功能的应用。如果搜索引擎告诉用户有很多结果与查询匹配，但是页面上只显示了一条（或者开头几条）结果，这就是字段折叠功能的体现。通常这种情况下页面会提供链接，以供用户点击展开，进而查看完整的搜索结果。如果没有对搜索结果进行折叠的话，就还能查看返回的额外文档的数量。

除了对搜索结果进行折叠、去除重复文档以外，Solr 的结果分组还提供了一些其他有用的功能。在许多情况下，Solr 的结果分组可视为更详细的分面操作形式。结果分组功能不仅能返回单独的分面块及每个值的数量，还能返回不同的值及其个数（与分面类似），以及包含了指定值的一些文档。分组与分面的区别之一在于，分组是在搜索结果里返回请求的分组。这意味着，分组及其值的排序都是基于查询中文档指定的顺序。Solr 可以按照字段值、函数或者查询执行分组功能，所以其应用范围很广。初次接触可能会觉得很复杂，本章会举几个例子来演示其强大且简单的功能。在此之前，先来区分一下有时会令人困惑的两个概念：结果分组及字段折叠。

## 11.1 结果分组 vs. 字段折叠

经常会被问到的一个问题是，为什么 Solr 的结果分组功能会有两个不同的名称——结果分组（Result Grouping）和字段折叠（Field Collapsing）？在一些搜索引擎中，字段折叠更为常见，具体是指删除重复值再返回结果集的行为。Solr 曾经开发过该功能的早期版本，其非官方的叫法是“字段折叠补丁”。后来该功能变得更加通用，更名为“结果分组”，想要表达其具备更多通用性的功能。

Solr 结果分组不仅可以对一个字段的重复值进行折叠并返回单一结果集，即字段折叠，还可以对单个查询返回多个结果集或者分组。因此，比起传统的字段折叠，“结果分组”这个名称可能更加名副其实。当必须删除重复文档时，自 Solr 4.6 起，Solr 提供另外一种实现方式（参见 11.7 节），能够提供更高效的字段折叠功能。不过这种方法也存在局限性，比如先于字段折叠的文档排序支持较弱，但在某些情况下可能也是有用的。

本章主要介绍 Solr 的结果分组的多种用法。首先从最常见的字段折叠功能入手，即删除搜索结果集中的重复文档。

## 11.2 忽略重复文档

假设你正在管理一家用户发帖卖东西的在线电子商务网站。在用户搜索商品时，尽管在结果中能看到同一件商品的成千上万个结果，但是针对单个商品只显示一份文档（连同该商品的数量一起显示）也许能提供更好的用户体验。这样会让搜索结果呈现多样性——当出现在结果顶部的多个相同商品并不是用户真正想找的，这是很必要的。比如，用户搜索 spider-man，假设最佳匹配的商品是 The Amazing Spider-man-2012，并且网站上有该商品的 100 个复制品。如果它们的内容一样，相关性得分也都相同，那么前 100 条结果都是该商品。相反，如果根据商品名

称对结果进行分组，要求每个组只出现一件商品，那么 The Amazing Spider-man-2012 就只会显示一次，用户也能看到其他的可能性结果，如 Spider-man-2002、Spider-man 2-2004 和 Amazing Spider-man Comic#2，详情见图 11.1。

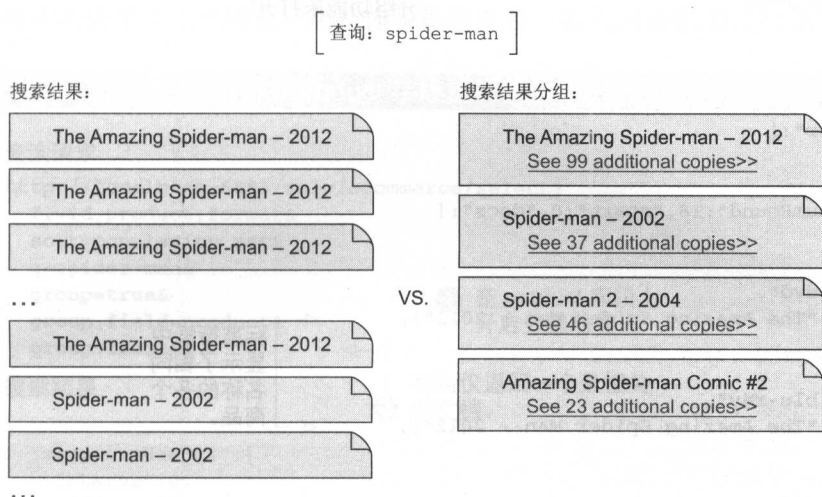


图 11.1 左边的标准搜索结果显示了重复文档；右边的分组搜索结果显示了不同的商品和结果折叠后的相同商品的数量

在大多数情况下，使用结果分组功能对搜索结果进行折叠能提供更好的用户体验。如图 11.1 左侧，在显示下一个不同结果前，默认将 The Amazing Spider-man-2012 显示 100 次，这会让那些想要搜索不同 spider-man 商品的用户感到非常沮丧和挫败。

为了实际演示该功能，随书源代码中已经添加了示例电子商务文档。下载 Solr 的干净（clean）版本之后，运行 Solr，使用代码清单 11.1 的命令加载示例文档。

#### 代码清单 11.1 为电子商务网站的示例文档构建索引

```
cd $SOLR_INSTALL/example/
cp -r $SOLR_IN_ACTION/example-docs/ch11/cores/ecommerce/ solr/ecommerce/
java -jar start.jar
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/ecommerce/update
➡ -jar post.jar ch11/documents/ecommerce.xml
```

当电子商务搜索引擎建立并运行起来之后，就可以实际演示 Solr 的结果分组功能了。代码清单 11.2 演示了关闭结果分组功能后，查询 spider-man 的标准搜索请求。



## 代码清单 11.2 包含重复文档的标准搜索

## 查询请求

```
http://localhost:8983/solr/ecommerce/select?
```

```
fl=id,product,format&
```

```
sort=popularity asc&
```

```
q=spider-man
```

基本关键词搜索，  
分组功能未打开。

## 搜索结果

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 2,
  },
  "response": { "numFound": 18, "start": 0, "docs": [
    {
      "id": "4",
      "format": "dvd",
      "product": "The Amazing Spider Man - 2012",
    },
    {
      "id": "5",
      "format": "blu-ray",
      "product": "The Amazing Spider Man - 2012",
    },
    {
      "id": "6",
      "format": "dvd",
      "product": "Spider Man - 2002",
    },
    {
      "id": "7",
      "format": "blu-ray",
      "product": "Spider Man - 2002",
    },
    {
      "id": "8",
      "format": "dvd",
      "product": "Spider Man 2 - 2004",
    },
    {
      "id": "9",
      "format": "blu-ray",
      "product": "Spider Man 2 - 2004",
    },
    {
      "id": "11",
      "format": "xbox 360",
      "product": "The Amazing Spider-Man",
    },
    {
      "id": "12",
      "format": "ps3",
      "product": "The Amazing Spider-Man",
    },
    {
      "id": "13",
      "format": "xbox 360",
      "product": "Spider-Man: Edge of Time",
    },
    {
      "id": "14",
      "format": "ps3",

```

标准搜索中  
显示了相同  
名称的多个  
商品。

```
    "product": "Spider-Man: Edge of Time"
  }
}
```

毫无疑问，这样的结果会产生极差的用户体验。因为结果将相同产品的不同格式（例如，蓝光光碟和 DVD）都以单条结果呈现了多次。开启结果分组功能之后，搜索结果呈现出多样化，用户体验得到改善，如代码清单 11.3 所示。

代码清单 11.3 对 product 字段进行折叠后的分组搜索结果

#### 查询请求

http://localhost:8983/solr/ecommerce/select?

```
fl=id,product,format&
sort=popularity asc&
q=spider-man&
group=true&
group.field=product&
group.limit=1
```

#### 搜索结果

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 3,
    "grouped": {
      "product": {
        "matches": 18,
        "groups": [
          {
            "groupValue": "The Amazing Spider Man - 2012",
            "doclist": { "numFound": 2, "start": 0, "docs": [
              {
                "id": "4",
                "format": "dvd",
                "product": "The Amazing Spider Man - 2012"
              }
            ]
          },
          {
            "groupValue": "Spider Man - 2002",
            "doclist": { "numFound": 3, "start": 0, "docs": [
              {
                "id": "6",
                "format": "dvd",
                "product": "Spider Man - 2002"
              }
            ]
          },
          {
            "groupValue": "Spider Man 2 - 2004",
            "doclist": { "numFound": 3, "start": 0, "docs": [
              {
                "id": "8",
                "format": "dvd",
                "product": "Spider Man 2 - 2004"
              }
            ]
          }
        ]
      }
    }
  }
}
```

① 在 product 字段上开启分组功能。

② 每组仅返回一个最相关的文档。

③ 分组结果有详细的返回格式。

④ 每个分组有一个 groupValue、numFound 和文档列表。

```

"groupValue": "The Amazing Spider-Man",
"doclist": { "numFound": 2, "start": 0, "docs": [
  {
    "id": "11",
    "format": "xbox 360",
    "product": "The Amazing Spider-Man"
  }
]},
{
"groupValue": "Spider-Man: Edge of Time",
"doclist": { "numFound": 2, "start": 0, "docs": [
  {
    "id": "13",
    "format": "xbox 360",
    "product": "Spider-Man: Edge of Time"
  }
]},
{
"groupValue": "Spider-Man Halloween Costume",
"doclist": { "numFound": 1, "start": 0, "docs": [
  {
    "id": "15",
    "format": "costume",
    "product": "Spider-Man Halloween Costume"
  }
]},
{
"groupValue": "Boys Spider-Man T-shirt",
"doclist": { "numFound": 1, "start": 0, "docs": [
  {
    "id": "21",
    "format": "shirt",
    "product": "Boys Spider-Man T-shirt"
  }
]},
{
"groupValue": "Amazing Spider-Man Comic #1",
"doclist": { "numFound": 1, "start": 0, "docs": [
  {
    "id": "22",
    "format": "paperback",
    "product": "Amazing Spider-Man Comic #1"
  }
]},
{
"groupValue": "Amazing Spider-Man Comic #2",
"doclist": { "numFound": 1, "start": 0, "docs": [
  {
    "id": "23",
    "format": "paperback",
    "product": "Amazing Spider-Man Comic #2"
  }
]},
{
"groupValue": "Amazing Spider-Man Comic #3",
"doclist": { "numFound": 1, "start": 0, "docs": [
  {
    "id": "24",
    "format": "paperback",
    "product": "Amazing Spider-Man Comic #3"
  }
]}
]}
]]]]

```

代码清单 11.3 中分组功能的几个属性值得注意。首先, 必须通过指定参数 `group=true` 开启分组功能 ❶, 另外还要指定对哪个字段进行分组 (如 `product` 字段)。将 `group.limit` 参数设置成 1, 表示每组里唯一的值只返回一个文档。如 11.4 节中, 将 `group.limit` 参数 ❷ 设置成大于 1 的整数也是有帮助的, 但是对于删除所有重复文档来说, 用户只想根据折叠的字段对每个唯一的值返回一个文档。

注意, 这里的结果格式也与 Solr 默认的结果格式 ❸ 有极大的差别。要将与搜索结果分组相关的所有信息展示给用户看, 需要更加详细的格式描述: 分组的字段名称 ❹, 定义每个字段的唯一词项 (`groupValue`), 以及折叠之前每组里的结果总数 (`numFound`)。

不幸的是, 对于解析两个单独的 Solr 结果格式来处理分组功能不太方便。不过如果只需要删除重复文档, 不需要所有这些额外分组信息的话, Solr 还提供了 `group.main` 参数, 将其设置成 `true` 的话, 该参数能将每组的结果合并成一个扁平化清单列表, 并且返回主要的结果格式。以下的代码清单 11.4 和代码清单 11.3 执行相同的查询, 不同之处在于这里将 `group.main` 参数设置成了 `true`。

#### 代码清单 11.4 将主要的搜索结果格式以扁平化分组结果的方式呈现

##### 查询请求

```
http://localhost:8983/solr/ecommerce/select?
  fl=id,product,format&
  sort=popularity asc&
  q=spider-man&
  group=true&
  group.field=product&
  group.main=true
```

将所有组的文档  
折叠起来, 放入  
主搜索结果中。

##### 搜索结果

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 8,
    "response": { "numFound": 18, "start": 0, "docs": [
      {
        "id": "4",
        "format": "dvd",
        "product": "The Amazing Spider Man - 2012",
        {
          "id": "6",
          "format": "dvd",
          "product": "Spider Man - 2002",
          {
            "id": "8",
            "format": "dvd",
            "product": "Spider Man 2 - 2004",
            {
```

折叠后  
的文档  
当前是  
唯一的。

```

    "id": "11",
    "format": "xbox 360",
    "product": "The Amazing Spider-Man"},
  {
    "id": "13",
    "format": "xbox 360",
    "product": "Spider-Man: Edge of Time"},
  {
    "id": "15",
    "format": "costume",
    "product": "Spider-Man Halloween Costume"},
  {
    "id": "21",
    "format": "shirt",
    "product": "Boys Spider-Man T-shirt"},
  {
    "id": "22",
    "format": "paperback",
    "product": "Amazing Spider-Man Comic #1"},
  {
    "id": "23",
    "format": "paperback",
    "product": "Amazing Spider-Man Comic #2"},
  {
    "id": "24",
    "format": "paperback",
    "product": "Amazing Spider-Man Comic #3"}]
  }}

```

使用 `group.main` 选项的最大缺点在于，不能获得每组里未折叠结果的总数。如果搜索应用在这方面并不是很在意，那么这可能是公平的交易，换来了不必处理两种不同的搜索结构格式的好处。使用该选项也不能获得简单格式的分组名称，但可以通过返回文档里的分组字段结果而得到（假定为单值字段）。使用 `group.main` 选项的另外一个缺点是，它仅支持请求单个组。

与标准搜索请求相比，使用结果分组执行字段折叠的另一个不足之处在于，执行速度会慢很多。所幸，从 Solr 4.6 开始，Solr 实现了更高效的字段折叠功能，该功能（参见 11.7 节）依赖于特殊的查询解析器（`CollapseQParserPlugin`）对文档进行折叠。这个功能会以标准的 Solr 搜索结果格式产生结果（就像分组功能里使用参数 `group.main=true`），而且也包含大量文档（这些文档都有用来折叠的唯一值）的索引提供了极大的性能提升。然而，如果使用 Solr 4.6 之前的版本，就只能如本节前面写到的那样使用速度更慢的结果分组功能了。

至此，我们仅仅指定了一个 `group.field=product` 参数，但是 Solr 支持返回多组结果。例如，可以同时设置 `group.field=type` 和 `group.field=format`，Solr 就会按照分组结果的格式返回两个组。然而，如果指定 `group.main=true`，Solr 将只返回在 Solr 的请求 URL 中指定的最后一组结果。

还有一种介于高级分组格式（默认）和 `group.main` 格式之间的选择——简单分组格式。通过指定 `group.format=simple`，返回多组结果（正如默认的高级分组格式那样），也能像 `group.main=true` 选项那样以扁平化清单列表方式返回每组请求的结果。实际上，从实现的角度来看，设置 `group.main=true` 参数使用的是 `group.method=simple` 功能，并能在主要结果清单中返回最后指定的组。代码清单 11.5 与代码清单 11.3 和代码清单 11.4 执行相同的查询，只是使用了简单分组格式。

### 代码清单 11.5 简单分组格式

#### 查询请求

```
http://localhost:8983/solr/ecommerce/select?
```

```
fl=id,product,format&
sort=popularity asc&
q=spider-man&
group=true&
group.field=product&
group.format=simple
```

启用简单分组格式。

#### 搜索结果

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 2,
    "grouped": {
      "product": {
        "matches": 18,
        "doclist": { "numFound": 18, "start": 0, "docs": [
          {
            "id": "4",
            "format": "dvd",
            "product": "The Amazing Spider Man - 2012",
          },
          {
            "id": "6",
            "format": "dvd",
            "product": "Spider Man - 2002",
          },
          {
            "id": "8",
            "format": "dvd",
            "product": "Spider Man 2 - 2004",
          },
          {
            "id": "11",
            "format": "xbox 360",
            "product": "The Amazing Spider-Man",
          },
          {
            "id": "13",
            "format": "xbox 360",
            "product": "Spider-Man: Edge of Time",
          },
        ]
      }
    }
  }
}
```

依然返回整体分组统计信息。

每个组的 `groupValue` 与 `numFound` 不再返回。

```

{
  "id": "15",
  "format": "costume",
  "product": "Spider-Man Halloween Costume"},
{
  "id": "21",
  "format": "shirt",
  "product": "Boys Spider-Man T-shirt"},
{
  "id": "22",
  "format": "paperback",
  "product": "Amazing Spider-Man Comic #1"},
{
  "id": "23",
  "format": "paperback",
  "product": "Amazing Spider-Man Comic #2"},
{
  "id": "24",
  "format": "paperback",
  "product": "Amazing Spider-Man Comic #3"}]
}
}
}
}

```

本节介绍了如何将查询结果折叠成不同的组，以便移除重复文档，也介绍了分组搜索结果能返回的三种格式：默认高级分组格式、简单分组格式及 `groups.main` 主要搜索结果中返回单一折叠分组。每种格式都在标准搜索结果格式的向后兼容以及描述指定分组的信息丰富性之间提供了折中方案。将搜索结果按照每个唯一的字段值折叠成单一文档的功能体现了传统的基于词项的字段折叠的精髓。

本章其余部分将介绍 Solr 的分组功能的更通用的高级用法。下一节首先介绍如何在单个查询中按组返回多个文档。

## 11.3 搜索结果中每组返回多个文档

Solr 分组功能可以按照各个唯一的字段值将搜索结果折叠成单个文档，但并不是 Solr 分组功能唯一可行的用例。返回前一节提到的电子商务搜索引擎例子，假设该实例不仅仅要删除重复文档，而且要确保从每个商品种类中返回的结果数不超过某个固定值。在 11.2 节中提到的搜索 `spider-man` 商品示例中，我们假设这样的情况：不返回折叠（去重）文档的扁平化清单列表，而是为每个产品类别返回三个文档，图 11.2 演示了该查询的执行情况。

代码清单 11.1 中索引的示例文档包含了可以用于分组的 `type` 字段，该字段请求每组最多返回 3 个文档，还可以请求每次最多返回 5 个组。代码清单 11.6 展示了该查询的执行情况。



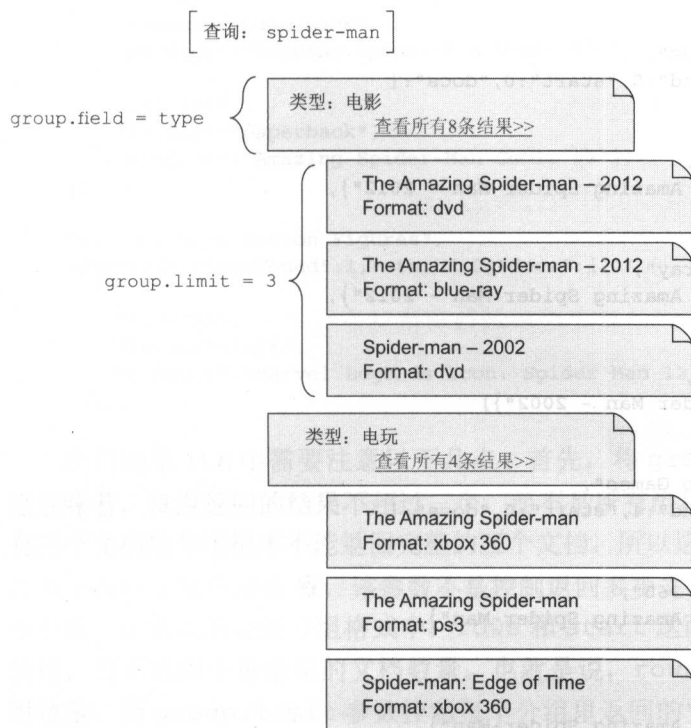


图 11.2 设置 group.limit&gt;1, 每组可以返回多个文档

## 代码清单 11.6 每组返回多个值 / 文档

## 查询请求

```
http://localhost:8983/solr/ecommerce/select?
q=spider-man&
fl=id,product,format&
sort=popularity asc&
group=true&
group.field=type&
group.limit=3&
rows=5&
start=0&
group.offset=0
```

- ① 每个组最多返回 3 个结果。
- ② 最多返回 5 个分组。

## 搜索结果

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 2,
    "grouped": {
      "type": {
        "matches": 18,
```

```

"groups": [{
  "groupValue": "Movies",
  "doclist": { "numFound": 8, "start": 0, "docs": [
    {
      "id": "4",
      "format": "dvd",
      "product": "The Amazing Spider Man - 2012"},
    {
      "id": "5",
      "format": "blu-ray",
      "product": "The Amazing Spider Man - 2012"},
    {
      "id": "6",
      "format": "dvd",
      "product": "Spider Man - 2002"}
  ]},
  {
    "groupValue": "Video Games",
    "doclist": { "numFound": 4, "start": 0, "docs": [
      {
        "id": "11",
        "format": "xbox 360",
        "product": "The Amazing Spider-Man"},
      {
        "id": "12",
        "format": "ps3",
        "product": "The Amazing Spider-Man"},
      {
        "id": "13",
        "format": "xbox 360",
        "product": "Spider-Man: Edge of Time"}
    ]},
    {
      "groupValue": "Clothing",
      "doclist": { "numFound": 2, "start": 0, "docs": [
        {
          "id": "15",
          "format": "costume",
          "product": "Spider-Man Halloween Costume"},
        {
          "id": "21",
          "format": "shirt",
          "product": "Boys Spider-Man T-shirt"}
      ]},
      {
        "groupValue": "Comic Books",
        "doclist": { "numFound": 3, "start": 0, "docs": [
          {
            "id": "22",
            "format": "paperback",
            "product": "Amazing Spider-Man Comic #1"},
          {
            "id": "23",

```

```

    "format": "paperback",
    "product": "Amazing Spider-Man Comic #2"},
  {
    "id": "24",
    "format": "paperback",
    "product": "Amazing Spider-Man Comic #3"}]
  }},
{
  "groupValue": "Action Figures",
  "doclist": { "numFound": 1, "start": 0, "docs": [
    {
      "id": "25",
      "format": "n/a",
      "product": "Marvel Legends Icon: Spider Man 12\" Action Figure"}]}
  ]}]}}}

```

代码清单 11.6 中需要注意以下几点：首先，将 `group.limit` 参数设置成 3，这意味着，每组返回的结果不超过三个，而不是所有组都包含三个结果<sup>①</sup>。因为没有三个文档的分组根本不能返回完整的三个文档，所以这个数字仅代表上限。其次，注意 `rows=5` 这个参数<sup>②</sup>，该参数不是控制返回多少文档，而是控制结果中返回多少个组。在默认的高级分组格式中，`rows` 和 `start` 这两个参数都和分组数量一起使用，而不适用于每组里的文档数量。也就是说，`rows` 参数控制结果中返回的分组数量，而 `group.limit` 参数控制每个分组里返回的文档数量。同样地，`start` 参数通过分组数量控制分页的 `group offset`，而 `group.offset` 参数通过每组里的文档数量控制分页大小。

代码清单 11.6 反映了 Solr 分组功能的最后一个重要方面：排序与分组的相互作用方式。从理论上讲，对所有分组进行排序的依据都是对组里匹配度最高的文档进行排序的顺序。也就是说，假定没有启用分组功能，所有文档都会按照排好的顺序出现（默认基于相关度得分）。如果得分最高的文档的 `type` 字段值为 `Movies`，排在第二的文档的 `type` 值为 `Comic Books`，排在第三的文档的 `type` 值为 `Movies`，排在第四的文档的 `type` 值为 `Clothing`，那么分组的排序应为 `Movies`、`Comic Books`、`Clothing`。

默认情况下，每个分组会根据文档首次出现的先后顺序对其进行排序。这意味着，因为 `Movies` 是最佳匹配的分组列表，并且要求在 `Movies` 的分组内显示三部电影，所以从严格意义上来说，第二部和第三部电影很有可能与查询不太相关（排序较靠后）。它们之所以在结果中出现的位置很靠前，是因为它们被提到了最佳匹配的分组 `Movies` 里面。鉴于这个原因，当使用 `group.main` 或者 `group.format=simple` 格式时，很少有人会将 `group.limit` 参数的值设置成大于 1。如果没有高级分组格式提供的结构来区分一个分组何时结束以及另外一个分组何时开始的话，排出来的结果顺序有可能会异常奇怪。

正如上面示例中提到的那样，针对一个字段值的分组能够提供有用的搜索功能，不过 Solr 的结果分组功能还能提供更多的可能性。下一节开始会讲解其他功能，首先是如何针对任意查询及函数进行分组。

## 11.4 按照函数和查询对结果分组

除了针对不同字段值的分组功能，Solr 还支持另外两种分组用法。第一种与针对字段的分组功能相似，不同的是它根据函数查询动态计算值进行分组。第二种是 Solr 的查询分组功能，该功能允许同时执行多个查询并且返回单独的结果集。

### 11.4.1 按照函数进行分组

基于函数的分组功能是借助 `group.func` 参数来完成的，本章不会讲解所有可能用到的函数（Solr 中所有的函数讲解请参见第 15 章）。代码清单 11.7 显示了按照函数分组之后的搜索结果。在此例中，该函数将搜索结果按照受欢迎程度分成三组（最受欢迎 =1，比较受欢迎 =2，最不受欢迎 =3）。

代码清单 11.7 按照函数分组的搜索结果

查询请求

```
http://localhost:8983/solr/ecommerce/select?
  fl=id,product,format&
  sort=popularity asc&
  q=spider-man&
  group=true&
  group.limit=3&
  rows=5&
  group.func=map(map(map(popularity,1,5,1),6,10,2),11,100,3)
```

根据函数对查询结果分组。

搜索结果

```
{
  "responseHeader":{
    "status":0,
    "QTime":3},
  "grouped":{
    "map(map(map(popularity,1,5,1),6,10,2),11,100,3)":{
      "matches":18,
      "groups":[{"
```

响应格式与字段分组类似。

```

"groupValue":1.0,
"doclist":{"numFound":2,"start":0,"docs":[
  {
    "id":"4",
    "format":"dvd",
    "product":"The Amazing Spider Man - 2012"},
  {
    "id":"5",
    "format":"blu-ray",
    "product":"The Amazing Spider Man - 2012"}]
}},
{
  "groupValue":2.0,
  "doclist":{"numFound":4,"start":0,"docs":[
    {
      "id":"6",
      "format":"dvd",
      "product":"Spider Man - 2002"},
    {
      "id":"7",
      "format":"blu-ray",
      "product":"Spider Man - 2002"},
    {
      "id":"8",
      "format":"dvd",
      "product":"Spider Man 2 - 2004"}]
}},
{
  "groupValue":3.0,
  "doclist":{"numFound":12,"start":0,"docs":[
    {
      "id":"11",
      "format":"xbox 360",
      "product":"The Amazing Spider-Man"},
    {
      "id":"12",
      "format":"ps3",
      "product":"The Amazing Spider-Man"},
    {
      "id":"13",
      "format":"xbox 360",
      "product":"Spider-Man: Edge of Time"}]
}}]}}}

```

函数计算值变为  
groupValue。

如你所见，从理论上讲，按照函数对结果进行分组与按照字段值来分组是一样的，唯一的区别在于分组的字段值是动态计算生成的。在此例中，所有受欢迎程度的值被映射到三个分组上（受欢迎程度 1 ~ 5 被映射到组 1；受欢迎程度 6 ~ 10 被映射到组 2；受欢迎程度 11 ~ 100 被映射到组 3）。函数是可以嵌套的，该例中嵌套了三个映射函数。这表明，如果你想通过结合多个函数来全面控制所有计算出来的值，是完全可行的。如果对于某些用例来说，函数分组功能的限制性太强，那么也可通过查询对结果进行分组，以便用户可以指定按照任意值进行分组。

## 11.4.2 按照查询进行分组

11.3 节提到过可以折叠搜索结果，目的是只返回与字段特定值匹配的文档。Solr 除了可以将预定义的字段值进行分组，还可以按任意查询进行动态分组。例如，对以顾客为中心的用户体验进行高度定制化，返回三个搜索结果集：在用户地理位置 50km 范围以内的商品，在顾客可接受价格区间之内的商品，以及顾客偏爱种类范围内的商品。因为 Solr 允许在同一个查询请求中发送多个 `group.query` 参数，所以可以将这些结果集以单个分组的形式返回。

这里使用本章的数据集来演示按查询进行分组的功能。首先来看三个请求查询分组：一个查询匹配所有的 movie，一个查询匹配任何可以与关键词 `game` 匹配的文档，一个查询针对指定产品 `The Hunger Games`。代码清单 11.8 展示了该查询的执行情况。

### 代码清单 11.8 按照多个查询进行分组的搜索结果

#### 查询请求

```
http://localhost:8983/solr/ecommerce/select?
```

```
sort=popularity asc&
fl=id,type,format,product&
group.limit=2&
q=*&
group=true&
group.query=type:Movies&
group.query=games&
group.query="The Hunger Games"
```

① 请求 3 个不同的查询分组

#### 搜索结果

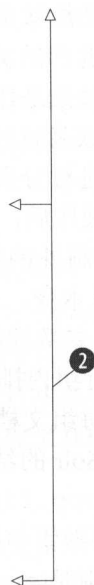
```
{
  "responseHeader": {
    "status": 0,
    "QTime": 1,
    "grouped": {
      "type:Movies": {
        "matches": 26,
        "doclist": { "numFound": 11, "start": 0, "docs": [
          {
            "id": "1",
            "type": "Movies",
            "format": "dvd",
            "product": "The Hunger Games",
            {
              "id": "4",
              "type": "Movies",
              "format": "dvd",
              "product": "The Amazing Spider Man - 2012"
            }
          }
        ]
      }
    }
  }
}
```

② 多个查询分组中会出现相同的文档。

```

"games":{
  "matches":26,
  "doclist":{"numFound":7,"start":0,"docs":[
    {
      "id":"1",
      "type":"Movies",
      "format":"dvd",
      "product":"The Hunger Games"},
    {
      "id":"2",
      "type":"Video Games",
      "format":"xbox 360",
      "product":"Dance Dance Revolution"}}
  ]},
  "The Hunger Games":{
    "matches":26,
    "doclist":{"numFound":2,"start":0,"docs":[
      {
        "id":"1",
        "type":"Movies",
        "format":"dvd",
        "product":"The Hunger Games"},
      {
        "id":"3",
        "type":"Books",
        "format":"paperback",
        "product":"The Hunger Games"}
    ]}
  ]}
}

```



该实例演示了三个关键之处：

- ❶ 可以请求从 solr 中返回多个组。这也适用于很多分组查询 (group.field、group.func 或 group.query)；只需一次请求，Solr 就可返回任意多个分组数。
- 查询分组是执行多个子搜索的一种方式。该例中，最初的查询是完全开放的搜索 (q=\*:\*)，该搜索可以在一次请求中执行尽可能多的查询，每次查询能返回单独的结果集。
- 尽管在分组结果集里一个文档只出现一次，但是在 Solr 的查询请求中运用了多个分组参数的话，每个分组结果集 ❷ 都能再次包含该文档，这一点很重要。这就像在同一个请求中正确运行多个搜索，因为如果相应参数 group.query 中的查询能与文档匹配的话，单独请求的查询分组也可以包含相同的文档。

由于 Solr 可以在一个请求中执行多个子查询，分组功能、搜索结果分页与文档排序在查询请求过程中就会发生有趣的相互作用。下一节将深入介绍它们之间的相互作用。

## 11.5 对分组结果进行分页和排序

运用了分组功能之后，搜索结果的结构会更加丰富，因此要对搜索结果进行分



页和排序时，分组功能就更加复杂了。在第 7 章中讲到了 Solr 使用 rows 参数能决定标准搜索查询中返回的文档数量。然而对结果分组时，还存在另外一层复杂性：你试图进行限制的对象是什么？是希望每组返回一定数量的文档，还是所有组返回一定数量的文档，或者返回既定数量的分组？类似的问题也存在于使用 start 参数对分组搜索结果进行分页和使用 sort 参数对分组搜索结果进行排序中。已经对搜索结果进行分组操作后，还要将其进行分页和排序到底意味着什么呢？

为了处理这种额外的复杂性，Solr 的分组功能将全局参数 rows、start 及 sort 也应用到分组本身。rows 参数决定返回分组的数量，starts 参数控制可用分组的分页，sort 参数控制如何对分组进行排序（基于组内的最佳匹配文档），这一点与文档如何进行组内排序是相对的。

如果需要增加每组文档的数量，将一个组内的文档进行分页，或者将不同组里的结果进行排序，Solr 的结果分组功能有特定的参数来控制这些功能。如 11.3 节中提到的那样，group.limit 参数指定每组返回结果数量的最大值，它执行的是 rows 参数在非分组搜索中的功能。group.offset 参数允许将一个组里的结果进行分页，它执行的则是 start 参数在非分组搜索中提供的功能。图 11.3 演示了在对分组搜索结果进行分页时，start、rows、group.limit 和 group.offset 参数之间的相互作用。

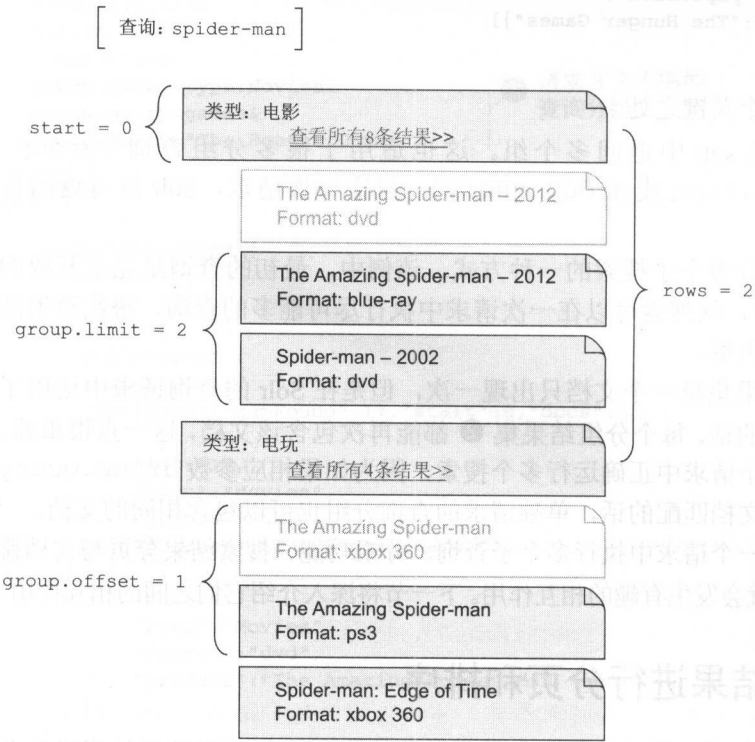


图 11.3 使用 start、rows、group.limit 和 group.offset 参数对分组搜索结果进行分页

在图 11.3 中, 每组返回的第一条结果颜色逐渐淡出, 表示该文档未被返回。该文档被忽略是因为将 `group.offset` 参数的值设置成了 1, 该参数能对每组里的结果进行分页。每个组仍然返回两个文档是因为将 `group.limit` 的值设置成了 2, 结果中返回两个组是因为将参数 `rows` 设置成了 2。因为将 `start` 参数设置成 0, 所以首先返回的是最佳匹配的分组。也可以通过将 `start` 参数设置为 1 直接跳到第二个分组上 (像处理每组中第一个文档那样)。

尽管已经使用过 `sort` 参数对它们进行初始排序 (初始排序决定了这些组各自出现的顺序), 你还可以用最后一个参数 `group.sort` 对不同组里的文档进行重新排序。这里可以实现双行程排序 (**two-pass sorting**): 一个阶段找到文档的相关分组, 另一阶段基于一些商业需求将组里的文档进行排序。图 11.4 展示了针对分组内的文档, `group.sort` 参数是如何与所有分组的默认排序进行交互的, 这里的默认排序依赖于 `sort` 参数。

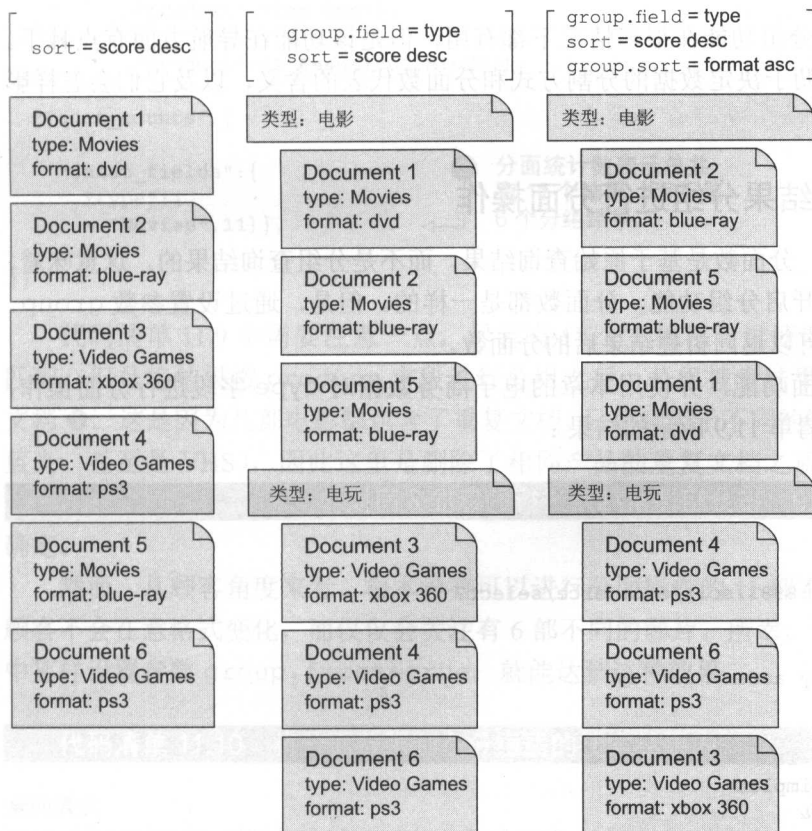


图 11.4 分组功能开启的排序示例。左侧栏显示根据文档得分的默认搜索排序; 中间栏显示组内文档仍使用相同排序规则, 根据分组得分的默认排序; 右侧栏显示根据分组得分的默认排序, 但使用 `group.sort` 参数对每组里的文档进行了重新排序

如你所见，在每组返回多个文档时，分组功能可以带来有趣的交互作用。在图 11.4 中，文档 5 从第 5 个位置点移动到了第 3 个位置，由于开启了分组功能后，该文档是最佳匹配分组里面排在最后的文档。一旦使用 `group.sort` 参数对每组里的文档进行重新排序，文档 5 的位置就会移动到更上面。但是分组排序规则仍然与 `sort` 参数排序保持一致，即基于每组内的最佳匹配文档排序。

学完本节，对分组请求中的分页、排序和对结果进行限制的最简单理解方法就是，把分组当成 Solr 返回的文档。就像在标准搜索请求中可以对文档进行分页、排序和限制那样，也可以对这些文档组进行分页、排序和限制。Solr 还提供了额外的分组参数 (`group.limit`、`group.offset` 和 `group.sort`) 帮助用户对用于展示的分组结果进行查询限定。

## 11.6 分组陷阱

尽管 Solr 的分组功能在很多情况下都有用，但是该功能在导航方面有点棘手。了解这些细节有助于决定数据的分割方式和分面数代表的含义，以及它们会怎样影响查询性能。

### 11.6.1 根据结果分组进行分面操作

默认状况下，分面数是基于原始查询结果，而不是分组查询结果的。这意味着，不管查询时是否开启分组功能，分面数都是一样的。但是，通过设置参数 `group.facet=true`，可以返回折叠结果后的分面数。

如果开启分面功能，并使用本章的电子商务数据对 `type` 字段进行分面操作，将会得到如代码清单 11.9 所示的结果：

代码清单 11.9 按分组搜索结果进行的标准分面操作

#### 查询请求

```
http://localhost:8983/solr/ecommerce/select?
  fl=product&
  group=true&
  sort=popularity asc&
  q=*&
  facet=true&
  facet.mincount=1&
  group.format=simple&
  fq=type:Movies&
```

```
facet.field=type&
group.field=product
```

分别启用分组与分面。

搜索结果

```
{
  ...
  "grouped":{
    "product":{
      "matches":11,
      "doclist":{"numFound":11,"start":0,"docs":[
        {
          "product":"The Hunger Games"},
        {
          "product":"The Amazing Spider Man - 2012"},
        {
          "product":"Spider Man - 2002"},
        {
          "product":"Spider Man 2 - 2004"},
        {
          "product":"Top Gun"},
        {
          "product":"A Beautiful Mind"}}
      ]}},
    "facet_counts":{
      ...
      "facet_fields":{
        "type":[
          "Movies",11]},
        ...
      ]}
    }
  }
```

① product 字段折叠后还有 6 个文档分组。

② 分面统计数表示总共有 11 个结果，而不是 6 个分组结果。

代码清单 11.9 中需要注意一点，尽管有 11 个文档都与过滤器 `type:Movies` 匹配，但是将结果按 `product` 字段进行分组之后，分组搜索结果中只返回了 6 个文档 ①。这是因为几部电影都包含了重复文档，只是对应于不同的影片格式（DVD、蓝光，甚至是 VHS），因此这里是删除了相同产品的重复文档之后的结果。第二个特征在于分面数 ② 仍然是根据文档的总数来确定的，而不是根据分组后的文档数来确定。

然而，从顾客角度来看，根本没有可以进行分面操作的 11 部不同的影片，因为顾客不会在意格式变化，而仅仅会关注有 6 部不同的影片。所幸，如代码清单 11.10 中那样设置参数 `group.facet=true`，就能达到这种效果。

#### 代码清单 11.10 将分面数限制在分组后的搜索结果内

查询请求

```
http://localhost:8983/solr/ecommerce/select?
  fl=product&
  group=true&
  sort=popularity asc&
```

```

q=*&
facet=true&
facet.mincount=1&
group.format=simple&
fq=type:Movies&
facet.field=type&
group.field=product&
group.facet=true

```

请求两个分面，  
一个统计分组数，  
一个统计文档数。

查询请求

```

{
  ...
  "grouped":{
    "product":{
      "matches":11,
      "doclist":{"numFound":11,"start":0,"docs":[
        {
          "product":"The Hunger Games"},
        {
          "product":"The Amazing Spider Man - 2012"},
        {
          "product":"Spider Man - 2002"},
        {
          "product":"Spider Man 2 - 2004"},
        {
          "product":"Top Gun"},
        {
          "product":"A Beautiful Mind"}
        ]}},
      "facet_counts":{
        ...
        "facet_fields":{
          "type":[
            "Movies",6],
          ...
        ]}
      }
    }
  }
}

```

查询匹配的  
文档总数是  
11 个。

文档分组  
一共是 6  
个。

分面值是 6，该数字表示  
分组内的文档数量。

按照某个字段进行分组，根据折叠后的分组进行分面，这样的功能可以很容易地用在前面提及的场景中。如果不想将 `group.facet=true` 参数设置用于所有分面中，可以通过指定 `f.<field>.group.facet=true` 有选择性地开启此功能。该参数里的 `<field>` 代表分面的字段，并且可以对其应用 `group.facet` 参数。另外需要注意的是，分面分组功能不能应用于多个请求组中，即在 Solr 的 URL 上请求多组参数。因此，要注意如果没有开启分面分组功能，那么该功能将只应用于第一个请求结果的分组上。

## 11.6.2 分布式结果分组

使用 Solr 的结果分组功能时需要重点考虑该功能如何与分布式搜索交互的问题。

与标准搜索不同，结果分组功能在分布式模式下不能完全起作用。相反，说它运行在伪分布模式下更正确一点。在分布式模式下，分组功能确实能返回汇总后的结果，但是仅仅是在每个 Solr 的内核上本地计算出来的分组的汇总结果。

为什么这一点很重要？如果计划进行分组的字段值随机分布在 Solr 的各个内核上，分组后结果中的文档数量就会不正确。如果按照商品查询中包含商品生产商名称的字段进行分组（查看该生产商所有不同的产品），分组的总数大概会和分布式搜索中搜索每个 Solr 内核的分组数量一样。当且仅当按照生产商名称将文档分割到不同的分片上时，才能够获取正确的分组数量，这是因为系统能确保每组只存在于一个分片上。如果计划在分布式模式下使用 Solr 的分组功能，并需要得到精确的分组数量（由 `group.ngroups=true` 参数返回），就一定要将此点牢记于心。如果没有按照计划那样，将数据按照分组的字段分割到各个分片上，那返回的分组数量仅仅是一个上限值。欲了解如何根据字段（如生产商名称）分割文档，以及使用自定义散列特征等内容，请参见 13.7.1 节。

除了这些数据分割限制以外，在分布式模式下几个分组参数目前也不能运行（`group.truncate` 和 `group.func`），如果考虑到数据会溢出内核，请谨慎使用这些功能。

### 11.6.3 返回扁平化列表

本章在 11.2 节中介绍过如何通过设置参数 `group.format=simple` 以简单分组格式（扁平化）（而不是默认的高级分组格式）返回分组结果。另外，还讲解了通过设置 `group.main=true` 参数可以以主要默认结果格式只返回结果中的第一个组（如未分组的查询）。尽管这两个选项都很有用，但是使用时也要仔细思考清楚：当搜索结果里没有高级分组格式提供的那些额外信息时，还能否得到想要的结果。没有高级格式，Solr 不能返回分组数量。如果使用分组功能对文档进行折叠，并且不使用高级格式的话，这意味着无法知晓能找到多少个不同的值。因为在应用程序中改变响应格式会比较具有挑战性，所以在初始应用程序开发过程中就应该仔细考虑要使用哪种格式。

### 11.6.4 按多值和分词字段进行分组

分组功能对于多值字段完全不起作用（在 `schema.xml` 文件中将参数设置成 `multivalued="true"`）；如果尝试这样做的话，程序会抛出异常。相反，根据分词字段（将文本分割成多个标记的字段）进行分组是可以返回结果的，但是返回的结果不太可靠，很大程度上也不太确定。具体地说，如果根据分词字段对结果进行分组，`groupValue` 只能代表每个文档的该字段里的标记，并且用户也不能选择使



用哪一个标记。因此，用户看不到针对每个标记的单独 `groupValue`，只能对正在进行分组的字段里看似随机的标记进行分组。Solr 试图将标记化字段看成一个单一的标记字段，这就使得分组功能不适用于包含了两个以上标记的任何字段上。就大多数用例来说，按字段进行分组的功能只适用于单一标记、非多值的字段上。

### 11.6.5 分组性能

虽然分组功能很强大，但是执行速度仍然比标准的 Solr 查询慢很多。对具有不同字段值的大量文档进行分组，分组查询会比非分组搜索请求花费的时间长得多。

为了提高分组功能的速度，可以使用 `group.cache.percent` 查询参数为分组查询启用缓存功能。此参数默认值为 0，将其设置为 1 到 100 之间的任何值都能开启此参数。Solr 的分组功能要求执行两个搜索，`group.cache.percent` 参数代表了，从第一个搜索开始，就能和其得分一起被缓存的文档占索引中所有文件的百分比，这是为了加快第二个搜索的速度。将此百分比设置得越高，分组查询消耗的内存就会越多，因此，为了用尽量小的数值达到最快速度，需要使用不同值进行实验。这是高级功能，它已被证明可以用来提高查询性能（不管是布尔、通配还是模糊查询）。不过使用时仍需要小心，因为它也被证明会降低简单查询的性能，如词项查询和匹配所有文档的查询（\*:\*）。

用户需要为搜索应用程序测试性能影响，包括是否开启分组缓存，但是相对于非分组查询来说，使用结果分组的查询很可能会降低性能。决定是否在用例中使用 Solr 的分组功能时，要将其性能影响纳入考虑范围。

如果只需要将结果折叠成每个字段值上唯一的文档，并不需要在折叠之前支持多值排序，那么用 Solr 4.6 之后的版本就能更高效地实现可用的字段折叠功能了，下一节会详细介绍。

## 11.7 使用折叠查询解析器进行高效的字段折叠

Solr 4.6 引入了折叠查询解析器（`CollapsingQParserPlugin`），该解析器不需要激活完整的结果分组堆栈，就能按照不同字段值将搜索结果折叠成单个结果。该功能的激活方式与其他查询解析器一样：

```
/select?q=*:*&fq={!collapse field=fieldToCollapseOn}
```

查询 `fieldToCollapseOn`，只会为字段里的不同值返回一个文档。在所有包含不同字段值的文档中，该文档的相关度得分是最高的。除了能返回最高得分的文档，查询 `field To CollapseOn` 可以在数值型字段或者函数查询中返回最低值或者最高值的文档。



```
/select?q=*&fq={!collapse field=fieldToCollapseOn min=numericFieldName}
/select?q=*&fq={!collapse field=fieldToCollapseOn max=numericFieldName}
/select?q=*&fq={!collapse field=fieldToCollapseOn max=sum(field1, field2)}
```

折叠查询解析器的最后一个功能是，借助 `nullPolicy` 以多种方式处理默认值的能力。如果文档的字段里有空值，而我们又需要按照这些字段折叠这些文档，这时就可以由 `nullPolicy` 来决定如何处理这些文档。`nullPolicy` 能提供三种选择：忽略、扩展和折叠。通过指定 `ignore` 的 `nullPolicy`，可以将所有带有空值的文档移除（不指定任何 `nullPolicy` 是默认设置）：

```
/select?q=*&fq={!collapse field=fieldToCollapseOn nullPolicy=ignore}
/select?q=*&fq={!collapse field=fieldToCollapseOn}
```

或者，如果想将缺失值分到一个分组里，可以指定 `collapse` 的 `nullPolicy`。该功能可以将所有包含 `null` 值的文档折叠成单一值。

```
/select?q=*&fq={!collapse field=fieldToCollapseOn nullPolicy=collapse}
```

最后，如果想将 `fieldToCollapseOn` 字段里带有重复值的文档折叠，同时又想返回带有 `null` 值的所有文档，则应该将 `nullPolicy` 设置为 `expand`。

```
/select?q=*&fq={!collapse field=fieldToCollapseOn nullPolicy=expand}
```

从理论上讲，折叠查询解析器返回来的结果与使用分组功能（`group=true`）以及指定 `group.main=true`、`group.limit=1` 和 `sort=score desc` 参数返回来的结果类似。（如果为折叠查询解析器指定最小值或者最大值，而不是指定字段参数，那么也可以根据数值型字段排序）。

使用折叠查询解析器存在一个局限：在折叠之前，搜索请求传入的 `sort` 参数对该解析器并不起作用。在对文档进行折叠前，分组功能首先要对文档进行排序，但是折叠查询解析器只注重单一因素（相关度得分或者最小/最大值）。也就是说，除了相关度得分或者最小/最大值以外，折叠查询解析器目前还不支持按照其他因素进行排序。自 Solr 4.7 起，折叠查询解析器不再支持根据函数计算出来的值对文档进行排序，这意味着，可以使用函数从技术上实现与排序的结合。

```
/select?q=*&fq={!collapse field=fieldToCollapseOn max=sum(product(field1,
1000000), cscore())}
```

在该例中，既然 `field1` 的值总是能比 `cscore` 函数高，所以我们先使用 `field1 desc` 再使用 `cscore desc` 对其排序是有效的。这是一种特殊的“折叠分数”函数，能在折叠之前获取文档的得分。如果需要在折叠之前进行排序，但又无法通过函数查询且进行折叠，则需要实施完整的分组功能。

## 11.8 本章小结

本章主要介绍 Solr 的结果分组 / 字段折叠功能。该功能包括许多查询选项：在查询时动态移除重复或者近似重复文档；确保结果来自不同的分组；或者甚至在一次请求中执行多个查询。结果分组也可以用来修改分面结果，在每次确定分面数量时排除重复的文档。本章还介绍了如何实现每组返回多个文档，以及如何按照字段、函数和查询进行分组。本章最后展示了在使用 Solr 的结果分组功能时会遇到的问题陷阱，以及分组功能的性能影响因素和分组结果可以返回的格式。

本书最后的 5 章涵盖了 Solr 的许多核心搜索功能。至此，你应该已经可以构建基于 Solr 的世界级搜索应用了。将搜索应用程序部署到生产环境中还需要其他工作。这是下一章的主题。

# 12

## 搭建Solr生产环境

---

### 本章要点

- 编译和部署 Solr 的分发版
- 监控和调试 Solr
- 跨服务器扩展 Solr 以处理大量内容和高并发查询
- 选择正确的配置（硬件、操作系统、JVM 和 Solr 缓存）

前面章节的大部分示例使用的都是比较小的数据集，只是为了展示 Solr 的核心功能及其原理。但是，在某一开发阶段，项目需要从原型进入能够处理大量查询请求和文档的生产环境。这也就意味着，我们还需要关注 Solr 功能之外的更多内容，这包括：服务器的配置（CPU、内存和操作系统）、服务器数量、服务器之间如何通信，处理负载时哪些 Solr 配置需要做相应修改、如何监控 Solr 的性能和调试 Solr 代码，以及 Solr 应用程序出现错误时如何修复等。另外，我们还需要考虑使用库文件（或者自己写代码）与 Solr 进行交互，执行查询，以及如何更有效地索引数据。本章将介绍这些部署 Solr 的生产环境的基本要素。

### 12.1 编写一份Solr的分发版

通常，官方新版本的 Solr（与 Lucene 一起）每年会发布多次。由于 Solr 是完全

开源的，因此用户也可以随时下载并改进 Solr 稳定版。Solr 的稳定版包含了所有开发者已提交的修改，因为还没有整合到官方发布版，所以稳定版的功能特性随时都会发生改变。

除了最新提交的修改，Solr 的 JIRA 页面 (<https://issues.apache.org/jira/browse/SOLR>) 还包含了世界各地的开发者贡献的补丁。一旦稳定版的管理者同意审查并认可开发者贡献的补丁，这些补丁便会被合并到稳定版中。因为这些代码都是公开的，并且可以免费使用，所以如果当前的官方发布版中没有想要的特性或者 bug 的修复，开发者可以自行开发适合自己的 Solr 的分发版。

因为大部分读者都会使用 Solr 的官方发布版（而不是自定义发布版），所以本章不会介绍如何获取 Solr 的不同分支的源代码，要了解这部分内容，请参见本书的附录 A，其中包括了以下内容：

- 如何应用 Solr 的补丁。
- 如何在 IntelliJ IDEA 和 Eclipse 等集成开发环境（IDE）中设置 Solr。
- 如何编写自己的 Solr 分发版。
- 如何在本地和远程服务器上调试 Solr 的代码库。
- 如何将包含自己对 Solr 的修改的补丁提交到开源社区。

无论你是想要在最喜欢的 IDE 中调试 Solr 的代码库，还是开发自己的 Solr 分支，附录 A 提供了快速集成 Solr 开发的详细教程。一旦结束了 Solr 的开发（或者下载最新的开箱即用的官方发布版），就可以开始 Solr 的编译和生产环境的部署了，这些将在本章的余下部分详细介绍。

## 12.2 部署Solr

Solr 在编译后形成一个标准的 Java Web 应用包（WAR 文件），该包能被部署到任何现代的 servlet 容器中。如果你对 WAR 文件在 Java servlet 容器中的运行原理还不熟悉，可以访问 [http://en.wikipedia.org/wiki/WAR\\_file\\_format\\_\(Sun\)](http://en.wikipedia.org/wiki/WAR_file_format_(Sun))，上面有对 WAR 文件与 Java servlet 容器运行原理的简介。运行本书的示例 Solr 应用程序（使用 start.jar）时，会启动一个名为 Jetty 的内置 Java servlet 容器来运行 solr.war 文件，当然也有很多用户选择将 Solr 部署到 Apache Tomcat 或其他 servlet 容器中。如果是要在 Java 应用程序中嵌入 Solr，也可以调用 Solr 库来集成 Solr。本小节将介绍编译 Solr 分发版，以及配置 Solr 的基础生产环境的过程。

### 12.2.1 编译自定义的 Solr 分发版

Solr 的官方分发版包含一个 \$SOLR\_INSTALL/dist/ 文件夹，该文件夹下有一

个 solr-\*.war 文件 (\* 代表 Solr 的版本号)。通常建议将 solr-\*.war 文件重命名为 solr.war。将 Solr 部署到 servlet 容器时, 该 WAR 文件和 Solr 的主目录 (\$SOLR\_INSTALL/example/solr/) 都是必需的。

### 编译 SOLR.WAR 文件

如果获取的是 Solr 的源码, 则 solr.war 文件需要手动编译。可以在 lucene-solr/solr/ 文件夹下面运行 ant dist 命令编译生成 solr.war 文件, 其中 lucene-solr/ 是 Lucene/Solr 的源码在本地的根目录。

```
cd lucene-solr/  
cd solr/  
ant dist
```

一旦上述命令运行成功, 你就只需要将生成的 lucene-solr/solr/dist/solr-\*.war 文件同 lucene-solr/solr/example/solr/ 文件夹 (如果修改了 Solr 的默认配置, 则为配置中指定的 Solr 根目录) 一起复制到 servlet 容器中。如果对该过程不熟悉, 可以访问 <http://wiki.apache.org/solr/SolrInstall> 查看 Jetty、Tomcat、GlassFish、JBoss、Resin、WebLogic 和 WebSphere 等主流 servlet 容器的配置指南。

### 在 JETTY 中部署 SOLR

Solr 中内嵌了 Jetty, 如果在 lucene-solr/solr/example/ 路径下执行 java -jar start.jar 命令启动 Solr 的示例应用, 那么也会同时启动内置的 Jetty 服务器。如果在运行 ant dist 命令编译生成 war 文件后执行上述命令, 则会导致 “solr.war 文件未找到” 的错误。

这是因为 Jetty 默认在 lucene-solr/solr/example/webapps/ 文件夹中寻找 war 文件, 而 ant dist 命令默认将生成的 solr.war 文件放在了其他文件夹中。要想使用 java -jar start.jar 命令, 可以手动将生成的 slor.war 文件复制到 lucene-solr/solr/example/ webapps/ 文件夹中, 或者执行 ant example 命令, 该命令在执行完 ant dist 后会将 WAR 文件复制到 lucene-solr/solr/example/ webapps 文件夹中。

```
ant example
```

将 lucene-solr/solr/example/start.jar 文件和 lucene-solr/solr/ example/solr/ 主目录一起复制到生产服务器中之后, 就可以成功地在生产环境中启动 Solr 了。

## 12.2.2 在应用程序中内嵌 Solr

除了将 solr.war 文件部署到 servlet 容器中, 还可以在 Java 应用程序中内嵌 Solr。如果需要给应用程序添加搜索功能, 且需要让 Solr 作为一个独立模块存在于

应用程序中，但又不想让外部应用程序或网络用户直接访问 Solr 服务，则非常适合这种内嵌的方式。在 Java 应用程序中启动内嵌的 Solr 的过程利用了 SolrJ Java 客户端，该客户端和 SolrJ 都将在 12.8.3 小节中介绍。无论将 Solr 内嵌在其他应用程序中还是将 Solr 作为独立的服务器，都必须确保有符合使用条件的硬件配置和系统配置，这些配置对应用程序的性能至关重要。下一节将介绍这些配置内容。

## 12.3 硬件和服务配置

通过扩展，Solr 可以对数十亿份文档的任意数量的查询请求进行处理，耗时从数十毫秒数千毫秒不等。但是，每秒处理数千份的查询请求和数十亿份的文档超过了单台 Solr 服务器的承受能力。对于大数据集，单台 Solr 服务器仅仅能存储几百万份文档，但是对于小数据集，单台 Solr 服务器可以在存储数亿份文档的同时保持合理的查询响应速度。尽管并没有统一的公式可以计算出特定用例下的最佳配置，但仍有一些通用的准则可供参考。

### 12.3.1 内存和固态硬盘

如果是在自己的服务器上（而不是在 Amazon EC2 或者 Rackspace Cloud Servers 等云服务平台上）运行 Solr 应用，那么提高应用程序性能的最好投资之一便是增加服务器的内存。与服务器的其他硬件相比，内存比较便宜，并且 Solr 对于内存的消耗非常巨大。即使使用云服务，也可以试着改用更大内存的产品类型，计算一下额外的金钱支出所换来的应用程序性能的提升是否值得。

Solr 进行分面、排序、索引文档和查询请求缓存时会消耗大量内存，因此需要确保给 Solr 分配了足够大的内存空间，以安全地存储上述数据结构。在 Solr 的管理控制台页面可以看到已分配的内存大小和当前可用的内存大小。

同查询速度一样重要的是，Solr 索引能否完全载入当前服务器未分配给 JVM 的可用内存空间中。如果 Solr 索引比服务器当前未分配给 JVM 的可用内存空间大，这就意味着，在处理搜索请求时需要进行硬盘寻道，以将索引文档加载到内存中，这会大大地降低应用程序的查询速度和查询吞吐量。

部分 Solr 使用者会购买固态硬盘（Solid-State Hard Drives, SSD）来抵消硬盘寻道的时间消耗。这确实能够加速硬盘寻道，但是最好不要将它作为首选优化方案。实际上，只要保证有足够大的内存空间，索引可以存储在内存中，而不必调用硬盘进行索引置换，就能够实现 Solr 最佳查询性能。如果需要更大的索引吞吐量，那么 SSD 可以提供更大的好处，因为索引的性能更多是受磁盘 I/O 限制。

在大部分现代操作系统（包括 Linux，它是最常见的部署 Solr 生产环境的操作

系统)中,当文档加载到内存,它将一直保存在内存中,直到系统要求回收这部分内存以作他用。保存内存中最近加载的文件直到所有可用内存都已消耗完毕(或者被缓存的文档最近不经常被访问,需要被释放以加载新的文档),这种机制允许应用程序在后续的请求中可以从内存中调用已缓存的文档。

因为 Solr 需要反复地访问相同的大块数据(Solr 的索引数据),所以操作系统的文件系统缓存机制对 Solr 服务器的总体性能就显得尤为重要。尽管许多 Solr 索引有数万亿字节的大小,超过了现今服务器中 RAM 的大小,但是我们可以管理 Solr 索引,控制它的大小总是小于 RAM 的大小。这样 Solr 在执行查询的时候就无须进行硬盘寻道,因为所有的文档索引都已经加载到了内存中。这种情况下,可以实现一个高效的内存搜索引擎,加速索引吞吐和查询处理。本小节要强调的重点是,尽管 SSD 性能很高,但是 RAM 通常是更合适的选择,而且,如果需要达到最快的处理速度,则应该考虑添加足够多的 RAM 以满足索引缓存的需求(以及任何分配给 JVM 以运行 Solr 的内存)。

### 12.3.2 JVM 设置

本书并不专门介绍 JVM 的垃圾回收,但是读者需要了解一些影响 Solr 性能的相关配置。首先,需要预先决定分配给 Solr 的 JVM 的内存大小,而不依赖 JVM 在需要时再去获取内存,例如:

```
java -Xms2g -Xmx2g -jar start.jar
```

或

```
java -Xms2048m -Xmx2048m -jar start.jar
```

在上述示例中,分配给 Solr 的 JVM 的内存最大值和最小值都为 2GB。Solr 实例可能需要 1GB 的内存,也可能需要 20GB 的内存,但是如果预先定义了最小(Xms)和最大(Xmx)的内存大小,就可以确保增加或削减可用内存的过程不会影响 Solr 的运行速度。在本章前面已经讨论过,不建议分配给 JVM 过多的内存空间,而应该只分配必要的内存空间来存储核心数据结构(缓存、Solr 内核和其他内存数据结构)和执行查询操作。这样操作系统能够在可用的内存空间中缓存 Solr 索引文件,这点是很重要的。如果分配了过多的内存,则 JVM 垃圾回收的时间会增加,因为一次要回收的垃圾更多了;同时,为了能够将 Solr 索引文件都加载到内存中,避免硬盘的寻道,需要使操作系统的文件系统缓存(前面的小节中已讨论过)处于内存匮乏的状态。通用的准则就是分配给 Solr 的内存空间比它所需的多一点点,而将其他的内存空间都留给操作系统。Solr 在管理页面(见后面小节中图 12.1)提供了内存的统计数据,可以帮助计算 Solr 实例消耗的内存大小。



## Solr 的垃圾回收

Java 的垃圾回收是一个令人头疼的过程。经常有人问哪种垃圾回收器更适合 Solr, 这是因具体部署环境而异的。如果要求在大多数时间里 Solr 都能达到最佳性能, 并且允许某段时间长期地暂停服务 (也就是说, Solr 无响应), 那么吞吐垃圾回收器 (Throughput Garbage Collector) 是不错的选择。64 位的服务器会默认开启它。

如果你追求的是长期稳定的性能, 即总体来说, 查询速度可能比较慢, 但是基本没有暂停服务 (Solr 无响应) 的情况或者暂停时间很短, 那么并发标记清除回收器 (Concurrent MarkSweep Garbage Collector) 可能是最佳选择。

```
java -server
-XX:+UseConcMarkSweepGC
-XX:+UseParNewGC
-XX:CMSInitiatingOccupancyFraction=80
-Xmx$JAVA_HEAP_SIZE
-Xms$JAVA_HEAP_SIZE
-jar start.jar
```

上述设置是对许多 Solr 使用者都适用的示例, 不过你还是应该查阅 JVM 提供商的文档, 查看其中垃圾回收器的可用选项, 根据实际环境中 Solr 服务器的性能指标来试验不同的垃圾回收器。许多垃圾回收器和可选设置的组合都可以用于优化 JVM 的垃圾回收过程, 但是好的垃圾回收配置是一个复杂的主题, 超出了本书的范围。如果垃圾回收器对你的 Solr 服务器的性能有重大影响, 可以查阅与 Java 垃圾回收相关的在线文档来了解更多信息。

### 12.3.3 索引切换

前面两小节主要介绍了内存和操作系统的文件系统缓存 Solr 索引文件的重要性。当新的文档被添加到 Solr 时, 索引将如何变化呢? 索引重构的过程对应用程序的性能有很大影响, 值得我们深入理解。

#### 增量索引

Solr 在内部利用 Lucene 创建文档的倒排索引。这个索引过程遵循增量索引原则, 即总是将修改添加到新文档中, 而不添加到之前已写的文件中。

从理论上讲, 这样的原则意味着一旦一份索引文档被创建了就不会再改变, 也就意味着如果要将一台服务器中更新后的索引复制到另一台服务器上, 只需要复制包含新增修改的新索引文件即可。因为索引通常可能有数十千兆或数百千兆字节, 所以增量索引在以下几个方面有优势:

- 因为之前的索引文件可以在多个新的索引版本之间共享, 所以减少了存储空间占用。

- 将一个服务器的修改复制到另一个服务器时只需要复制比较小的新增文档。
- 无论新版本的索引何时被提交，操作系统都缓存了大部分的索引（旧索引）。

但是，增量索引过程也要付出一些代价。最主要的代价就是旧索引文档永远不会被更新，因而任何文档更新和删除实际上都要占用一定的额外空间（因为它们需要重新被写到新的索引文档中）。因此，如果重复地将同一份文档发送给 Solr，会观察到索引的大小在持续增长，旧版本和新版本的文档都被保留在了索引中。当删除文档时，会有一份黑名单告诉 Solr 在索引时要忽略该文档的旧版本，但是旧版本仍然存在于之前的索引块中。如果重新索引所有内容（在一个包含旧内容的 Solr 实例中），即使大部分新索引的内容都没有发生改变，仍然可以预见索引的规模在这个过程中会大幅度增加。

上文提到旧索引文档永远不会被更新，这里进一步对比说明。在许多更新请求被提交之后，Solr 中存在许多索引块，需要将它们合并成新的索引块。从技术角度看合并过程并不会修改旧的索引块，而是创建新的索引块，使旧的索引块失效。当旧的索引块文件不再被 Solr 引用的时候，它们会被删除以释放磁盘空间。

正如在第 5 章中提到的，为了让新添加的文档能够被索引，必须要发生一次提交操作。硬提交操作启动一个新的搜索器，以取代当前正在运行的搜索器，新搜索器会创建新的索引（旧的索引块加上新增索引块）。

### 索引切换和缓存预热

当一次硬提交操作发生时，Solr 会创建一个新的搜索器，新的搜索器会引用索引中的旧索引块，以及所有新增的索引块。因为 Solr 可能持续接收到新的搜索请求，所以需要保证无论何时生成新索引都不会对 Solr 的查询体验产生影响，这一点很重要。

Solr 通过并行运行两个搜索器来实现新旧索引的无缝过渡。旧的搜索器（仅仅引用了旧索引块）继续接收用户请求，而新的搜索器（引用旧索引块和所有新增索引块）在后台被加载。

因为 Solr 利用缓存存储历史查询、过滤器、字段值及其他数据来加快搜索，所以当新的搜索器开始处理查询请求时，需要保证这些缓存继续工作。不幸的是，缓存都绑定了特定的索引版本，这意味着新搜索器必须要花时间利用旧搜索器的缓存数据预热新的缓存。需要解释的是，每当一个提交操作发生时，都会创建一个新的搜索器，并预热新的缓存。对新搜索器利用中的缓存总大小和缓存对象数目的配置都存储在 `solrconfig.xml` 文件中，每个缓存的定义如下：

```

<fieldValueCache class="solr.FastLRUCache" size="500" initialSize="50"
  autowarmCount="500" />
<filterCache class="solr.FastLRUCache" size="500" initialSize="500"
  autowarmCount="250" />
<queryResultCache class="solr.LRUCache" size="100" initialSize="100"
  autowarmCount="0" />
<documentCache class="solr.LRUCache" size="500" initialSize="500"
  autowarmCount="0" />

```

在上述的每个示例中，缓存大小的最大值都在 100 到 500 之间。在某些使用情况下，可能需要更大的缓存大小，尤其是当分配给 JVM 的内存很大，并且缓存命中率（详见 12.9.2 节）随着缓存大小的增加而线性增长时。但是，缓存的大小越大，新搜索器需要预热的缓存对象也就越多。

尽管设置一个较大的缓存没有什么问题（如果有足够的内存空间），但是在设置 autowarmCount 的时候则需要考虑实际情况。autowarmCount 的值越大，预热一个新搜索器所需的时间就越长，因此新添加的文档被索引的时间和提交操作执行的时间也就越长。

特别地，对于 filterCache，如果有许多非常耗资源的过滤器，可能需要数分钟甚至是数小时（极端情况下）才能完成一个新搜索器的预热，这对于许多搜索应用来说都是不实际的。但是，如果将 autowarmCount 的值设置得过低，可能会对新搜索器的初始性能产生负面影响，并且可能在执行复杂查询请求（如第一次进行分面）时造成应用程序的中断响应。因此，为获得合理的 autowarmCount 设置，优化新搜索器中的缓存效率及预热时间，你需要根据试错情况不断调整缓存大小。

## 索引块合并与优化

第 5 章中介绍过，当索引块变得很大时，它们会自动被合并（移除已被删除的文档和重复的文档副本）以压缩索引大小。是否及如何合并索引块由 solrconfig.xml 文件中定义的合并策略决定。合并调度器（Merge Scheduler）和合并策略（Merge Policy）决定何时以及如何合并索引。

```

<mergeScheduler class="org.apache.lucene.index.ConcurrentMergeScheduler">
  <int name="maxMergeCount">4</int>
  <int name="maxThreadCount">4</int>
</mergeScheduler>

<mergePolicy class="org.apache.lucene.index.TieredMergePolicy">
  <int name="maxMergeAtOnce">10</int>
  <int name="segmentsPerTier">10</int>
</mergePolicy>

<mergeFactor>10</mergeFactor>

```

TieredMergePolicy 是 Solr 中默认的合并策略，并且通常也是最有效的。maxMergeAtOnce 和 segmentsPerTier 这两个参数决定了索引块何时被合并。

`segmentsPerTier` 设置决定了在 Solr 开始合并索引块之前，可以创建多少个索引块；`maxMergeAtOnce` 设置决定了一次最多有多少索引块可以被合并到新索引块中，默认情况下，这两者的值都被设为 10。如果 `TieredMergePolicy` 的值也未被指定，则它的值将被设为单独定义的 `mergeFactor` 参数的值（默认值也是 10）。

通常情况下，`mergeFactor` 的值（或者 `maxMergeAtOnce` 和 `segmentsPerTier` 选项值）越大，Solr 索引文档的速度越快，因为索引块合并的次数少了很多。大量索引块会降低搜索的速度，因为需要查找更多的索引文件，而且每份索引文件可能包含尚未合并的旧版本文档和已删除文档。`mergeFactor` 的默认值 10 可以很好地适用于大多数使用情况，但是有时开发者还是需要增大或减小这个值，这取决于开发者更关心的是索引的速度还是搜索的速度。

某些使用情况中，可能需要初始化设置一个 `optimize` 参数将所有文档合并成为一个索引块。例如，如果仅仅需要对所有数据执行一次性的索引，为了使得搜索器在索引上运行得尽可能地快，可以在完成索引之后对索引进行优化，将其合并成一个索引块。除此之外，如果比较频繁地对索引执行大量的更新操作（例如每天都更新大部分的文档），那么建议在用户搜索请求较少时进行周期性优化，这样可以释放一些被旧文档的副本占据的存储空间。如果要优化整个索引，可以从 Web 客户端将包含 `<optimize/>` 标签的 POST 请求发送至 `/update` 处理器。如果手动进行优化，可以考虑换用更方便的 GET 请求的 `stream.body` 参数来发起优化请求，例如：`http://localhost:8983/solr/collection1/update?stream.body=<optimize/>`。

需要记住 `optimize` 操作非常消耗资源——该操作会按行重写所有的索引——并且可能需要花费较长的时间并消耗大量系统资源。如果索引块的合并策略设置得足够强有力，则可能不再需要优化索引，因为索引块会被频繁地合并，这合理地清除了索引中堆积的垃圾数据。因为在 `optimize` 过程中，索引会整体被重写，所以如果操作系统没有足够的 RAM 空间同时容纳新索引和旧索引，查询速度就会受到影响。如果索引分布于同一台服务器的多个 Solr 内核中，可以考虑每次仅仅优化一个 Solr 内核，这样有利于减小优化对查询速度的影响。执行 `optimize` 操作之前，需要权衡优化过程对查询速度的潜在负面影响以及优化结束带来的好处。

下一小节介绍操作系统级别的影响 Solr 服务器速度和稳定性的设置。

### 12.3.4 实用 Solr 系统配置技巧

Solr 的开发语言为 Java，也就意味着它可以运行在各种操作系统之上。实际中大部分生产环境中的 Solr 集群（以及生产环境中的服务器）都是运行在 Linux 操作系统之上的。本小节将介绍一些提高 Linux 环境下 Solr 系统性能的实用技巧。

## 自动预热操作系统的文件系统缓存

正如在 12.3.1 小节中介绍的, 如果服务器的 RAM 足够大, 大到可以容纳整个 Solr 的索引 (包括分配给 JVM 的内存), 那么就可以极大地提高 Solr 的性能, 因为这样可以利用操作系统的文件系统缓存第一次从硬盘加载到内存的文件。但是这种方式仍然需要初始化地将所有索引文件加载到内存中, 也就是说, 如果在系统重启 (或因为长时间的延时, 其他文件被加载到内存中, 替换了 Solr 索引) 之后, Solr 的运行速度会比通常情况下慢很多——尤其是用户请求了一份从未被请求过的文件时——直到最终将所有索引文件都加载到内存中。

针对 Solr 索引第一次访问时延迟加载而导致的应用程序响应速度变慢的问题, 有一个不错的解决办法: 在 Solr 启动之前使用下列命令预加载所有索引文件。

```
find $SOLR_INSTALL/example/solr/*/data/ -type f -exec cat {} \; > /dev/null
```

如果使用默认的目录结构存储每个 Solr 内核的索引 (\$SOLR\_INSTALL/example/solr/\$CORE\_NAME/data/), 则该命令会将文件夹中的所有索引文件逐份从磁盘加载到每个 Solr 内核中。如果不是默认目录, 则需要对上述命令中的路径做相应修改。对于大规模的索引 (数万千兆级别) 而言, 将所有索引文件预加载到操作系统的文件系统缓存的过程可能需要数分钟才能完成, 但是如果在启动 Solr 之前运行该命令, 则有利于加快搜索器的运行, 尤其是在刚刚重启了服务器的情况下。

为了测试这种方法是否有效, 可以运行上述命令两次: 一次在系统刚刚重启之后, 另一次在系统第一次运行时。你会发现第二次运行命令时, 速度明显快很多 (几乎是瞬时完成), 证明在第二次测试时, 文件是直接从内存加载的, 这也解释了为什么这种方式能够在第一次启动 Solr 时对 Solr 的性能有很大帮助。

记住, 如果 Solr 索引比空闲内存的空间要大, 执行上述方法并不会使得所有索引文件都处于内存中。实际上, 如果已经有了正在运行的 Solr 实例, 然后又执行上述方法将搜索索引文件加载到内存中, 那么很有可能会把内存中某些更重要的文件置换出去。总而言之, 如果有足够多的内存, 可以随意使用上述方法, 否则, 请慎重。

## 增加可用的文件描述符

由于 Lucene 增量索引过程的性质影响, 一个索引可能由数百份索引文件组成, 可能它们都需要在同一时间被打开。如果 Solr 中有许多索引, 也就意味着 Solr 可能有数千份索引文件, 这些文件需要保持在任意时刻都是打开状态并能够被搜索。但是同时打开这么多文件可能超过了一些基于 UNIX 的操作系统默认的上限值, 会导致 Solr 因无法打开更多的索引文件而崩溃。Solr 使用相对于最大文件描述符数量 (可被同时打开的最大文件数量) 的值, 该数量可以在 Solr 的管理页面的主面板中看到, 如图 12.1 所示。

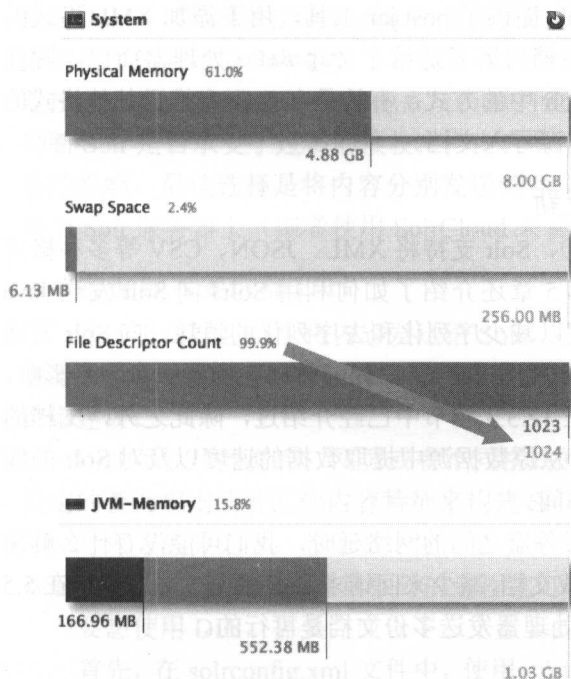


图 12.1 Solr 管理页面显示了文件描述符使用情况以及其他一些系统资源信息

从图 12.1 可以看出，Solr 打开的文件（大多数是多个 Solr 内核的索引文件）数目已经达到了系统文件描述符的上限，也就意味着 Solr 很有可能即将崩溃。想要修复这个问题很简单，使用 `ulimit -n` 命令即可。`ulimit -n` 命令将会告知用户可用的文件描述符数目。为了可以更加安全地运行 Solr，增加该上限，指定一个新的上限值即可。

```
ulimit -n 100000
```

许多系统都默认将文件描述符的上限设置为 1024，但是因为每个 Solr 索引包含了数百份索引文件（甚至可能是数千份，这取决于你的 MergePolicy 设置），可能将上限值增大为 100 000 或更大才比较合适，尤其是在服务器上运行多个 Solr 内核时。可以通过修改系统配置文件（如 `/etc/security/limits.conf`）来设置文件描述符上限的永久值（运行上文中的命令只能将修改应用于当前会话）。在配置的过程中需要确保新的文件描述符上限值足够大，以保证 Solr 永远无法达到该限制值。

## 12.4 数据获取策略

到目前为止，本书介绍了一种将文件添加到 Solr 中的方法：通过 HTTP 将文档



发送给 Solr 的 /update 处理器。Solr 提供了 post.jar 工具，用于添加 XML 格式的 Solr 文档。实际上，post.jar 只是将文档内容发送给了 /update 处理器而已。除此之外，还有很多其他将文档添加到 Solr 中的方式，有的是向 Solr 中推送其他格式的文档，有的是让 Solr 自己从其他数据源导入文档。

### 文档格式、索引时间和批量更新

正如第 5 章介绍的（参见 5.5 节），Solr 支持将 XML、JSON、CSV 等多种格式的文件发送到 /update 处理器。第 5 章还介绍了如何利用 SolrJ 向 Solr 发送 Java 二进制格式的文档，采取这种格式可以减少序列化和去序列化的消耗。向 Solr 发送文档时需要记住很重要的一点，即 Solr 索引文档所消耗的时间会受到很多因素影响。影响 Solr 索引内部配置的部分因素在 12.3.3 小节中已经介绍过，除此之外，文档的输入格式、网络调用数、网络延时、从原数据源中提取数据的速度以及对 Solr 的线程请求，都会影响到 Solr 索引的总时间。

对于发送内容的服务器和 Solr 服务器之间的网络延时，我们可能没有什么解决办法，但是可以在一次请求中批量加载文档，减少来回请求 Solr 的总次数。我们在 5.5 节中介绍过，一次性地向 /update 处理器发送多份文档是可行的。

```
<add>
  <doc>
    ...
  </doc>
</doc>
...
</doc>
...
</add>
```

用批量发送取代每份文档单独发送会在很大程度增加 Solr 的索引吞吐量。因为这样可以移除额外的网络延时，并且允许 Solr 一次性做更多的工作。通常情况下，一次向 Solr 发送数百份中型或大型文档是很合理的，并且对于小文档来说，一次发送数千份也是很符合实际的。你可以根据文档的大小和 Solr 处理文档添加请求的时间来反复试验批量更新文档的数量，决定应用程序的最佳阈值。

除了批量发送文档，还要注意选择合适的文档格式，因为这会对 Solr 的索引吞吐产生重要影响。解析 XML 需要消耗较多的资源，意味着选择 Java 二进制或 CSV 等易于解析的文档输入格式会在很大程度上减少索引时间。

最后，任何向 Solr 发送内容的过程都需要克服一些基本的性能瓶颈：其他线程忙于准备处理请求导致没有足够多的线程发送文档，或者是 Solr 一直在等待接收内容而没有创建索引。索引过程的一个最大的性能瓶颈便是从外部数据库中加载内容，并将它转化为 Solr 文档；在许多情况下，索引进程任由自己在进行这类处理时陷入



瓶颈状态，甚至都不会通知 Solr，这会导致 Solr 几乎停止工作。简而言之，大部分应用程序都会远在 Solr 的索引吞吐饱和之前就陷入性能瓶颈之中。因此，需要随时监控 Solr 服务器的 CPU 和其他系统资源的使用情况，在觉得 Solr 索引运行太慢时，判断 Solr 是否承受了过多的接收文档内容的压力。如果 Solr 确实受到了接收文档内容的影响，最佳选择是将内容分别发送到多个 Solr 内核之中，并将更新请求分配到多个 Solr 服务器上（或者使用 SolrCloud 来管理这个过程，详见第 13 章）。

### 数据导入处理器 DIH

除了手动发起请求将内容发送给 Solr，还可以利用数据导入处理器从外部数据源导入数据。10.1.1 节曾使用 DIH 导入 13 000 篇 Wikipedia 文章来测试不同形式的查询建议。

DIH 可以从数据库、网页或者不同格式的文本文件中导入数据，并且可以根据指定的查询规则或者匹配内容特征来构建 Solr 文档。

导入方式分为导入整个数据集（索引整个文档集合）和增量导入（导入上次导入后的修改），后者不用开启额外的进程就可以将数据导入 Solr 中。

要想使用 DIH，需要完成以下三个重要的配置步骤。

首先，在 `solrconfig.xml` 文件中，使用 `<lib>` 加载 DIH 的 JAR 文件和依赖文件。

```
<lib dir="../../../contrib/dataimporthandler/lib" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-dataimporthandler-.*\.jar" />
```

然后，在 `solrconfig.xml` 文件中，定义 `/dataimport` 请求处理器。

```
<requestHandler name="/dataimport"
  class="org.apache.solr.handler.dataimport.DataImportHandler">
  <lst name="defaults">
    <str name="config">data-config.xml</str>
  </lst>
</requestHandler>
```

最后写一份数据导入配置文档来配置 DIH。当在 `solrconfig.xml` 文件中定义 DIH 时，需要在示例请求处理器的配置中指定数据导入配置文档所在的路径，本章将该配置文档命名为 `data-config.xml`，路径名为相对于所在 Solr 内核中的 `conf/` 文件夹的相对路径。此文件包含所有外部数据源的引用信息，这些来源的数据都会被拉取。另外，此文件还包含了如何将这些数据记录转换成 Solr 文档的步骤。DIH 支持多种类型的外部数据源和复杂的内容转换逻辑。在数据导入时，支持对操作数据的任意脚本代码运行。这部分内容超出了本章范畴，数据导入处理器的详细指南参见 <http://wiki.apache.org/solr/DataImportHandler>，那里介绍了如何自动导入数据。

在本书的部分章节中，已经使用过或者还将使用大量有趣的文档集来进行实验。本章引用两个大型和免费的文档集：Wikipedia 和 Stack Exchange 的数据集。为了展

示数据导入处理器的强大，我们在附录 C 中给出了详细的指导和示例数据导入配置，以帮助读者快速地配置和运行 Solr，导入上述数据集。

一旦定义了 data-config.xml 文件，使之能够正确地导入数据并将其转换为文档，就可以点击 Solr 管理页面中的 Dataimport 按钮，开始导入数据并观察现阶段的导入状态了。图 12.2 展示了一个正在导入数据的示例。

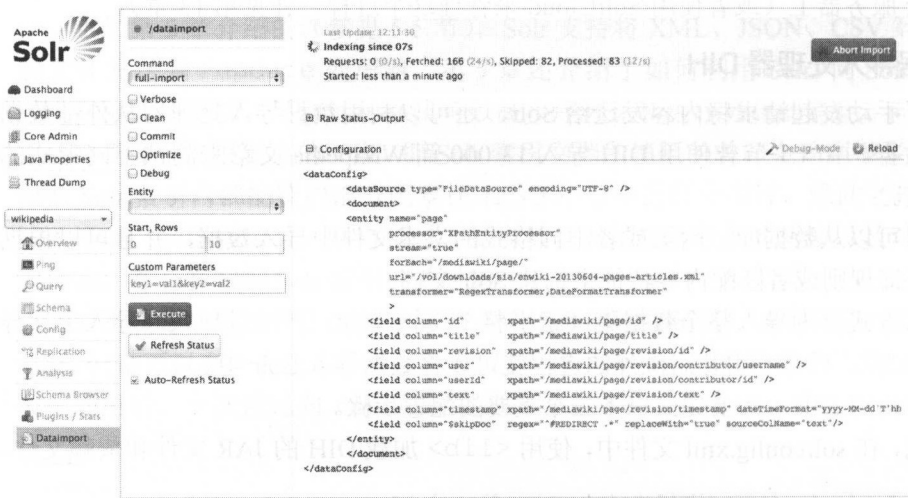


图 12.2 当前数据导入进度

在图 12.2 中，可以看到当前数据导入的状态，包括目前已经处理的记录条数。在左边，有导入新数据的选项（全导入或增量导入，从下拉框中选择），在右边，可以看到当前的 data-config.xml 配置、当前或者上一次导入的状态，以及放弃当前导入的选项。如果想更多地尝试将数据集直接导入到 Solr 的过程，可以参考附录 C 中的示例导入 Wikipedia 或 Stack Exchange 的数据集。

### 利用 Solr Cell 从非文本文档中抽取文本数据

除了从数据库、网站和文件中导入文本数据，Solr 还可以从非文本类型的文档中提取数据，如 PDF、图像、Word 文档、Excel 表格、PowerPoint 文档等其他常见文档格式。这项功能体现在 /extraction 请求处理器中，在 Solr 的示例配置中默认启用了该请求处理器。这项功能在 Solr 中称为 Solr Cell，源于 Solr 内容抽取库（Content Extraction Library）。这项功能利用 Apache Tika (<http://tika.apache.org/>) 从各种类型的文件中提取信息，可以在 Solr Cell 维基页面 (<http://wiki.apache.org/solr/ExtractingRequestHandler>) 查看配置选项。

读到这里，你应该清楚了 Solr 为内容处理提供了许多选项——通过 REST API 接受不同格式的内容，从数据库、网页、文件中自动导入数据，利用 Solr Cell 从非

文本文档中提取信息。但是一旦开始索引所有内容，一段时间后，单个 Solr 内核或单台服务器中就会索引太多的内容。也有可能用户查询请求太多，超过了单台服务器的承受范围。下一节将介绍如何通过分片和复制来扩展 Solr，以处理大量新增内容和用户查询。

## 12.5 分片和复制

第 3 章（见 3.4.2 小节）介绍了 Solr 中分片和分布式搜索的概念。Solr 允许创建多个索引，每一个索引背后都是一个 Solr 内核。可以将内容切分对应到不同的 Solr 索引中（称为分片），同时创建任何数据切片的多个副本（称为复制）。第 3 章已经介绍了如何在多个不同的 Solr 内核中执行分布式搜索，本节将会介绍如何决定 Solr 集群所需的分片和副本的数量。分片和副本将在第 13 章中同可以自动管理分片的 SolrCloud 一起被重点介绍。

### 12.5.1 分片策略

如果 Solr 索引的文档太多，超出了单台服务器的负载能力，分片就显得十分有用了。图 12.3 展示了一个简单的 Solr 集群分片，该集群基于简单的文档 id 字段哈希值的模运算结果来切分文档。

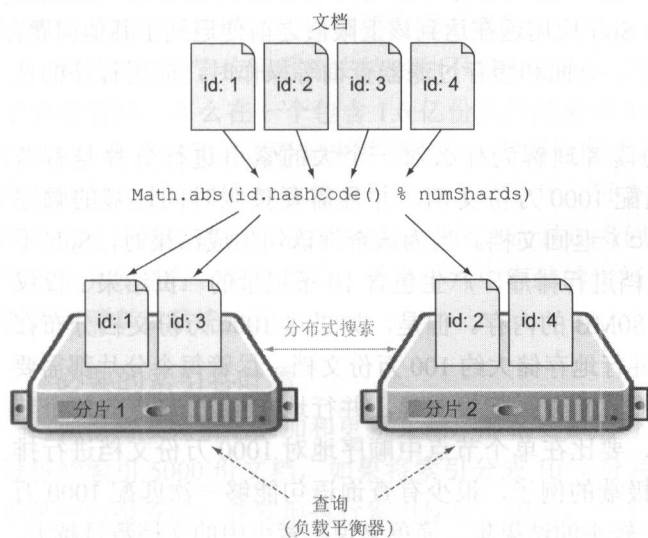


图 12.3 根据文档 ID 对两个分片进行文档分区。每个文档只出现在一个分片上，对任一台服务器的查询都需要对两个分片进行分布式搜索，以便搜索到所有文档

在图 12.3 中，因为每份文档仅会出现在其中一个分片之中，所以要想搜索所有

文档，就必须在两个分片中执行分布式搜索。如果对利用 shards 参数在多个分片中执行分布式搜索还不熟悉，可以参见第3章（3.4.3 小节）。

在设置 Solr 集群时，需要决定文档集的分片数量。根据索引的大小，可以设置一个或者数百个分片。如果一个文档集只有一个分片，则不需要考虑索引分片。本小节提供了决定所需分片数量的指导，但是因为要考虑的因素太多，所以需要根据实际数据进行测试，得出最佳值。

Solr 的分片数量与容错性无关。但分片在文档集增大时很有帮助。在详细介绍如何决定 Solr 分片的数量之前，应当了解，使用 Solr 4.3.1，你可以很容易地将 Solr 中一个已有的分片切分成两个更小的分片，因而不受初始值的限制。但是，因为分片切分旨在解决索引随时间推移自然增长所带来的问题，所以我们并不希望随意选择一个初始值。总体而言，在决定分片数量时主要有以下 5 个因素需要考虑：

1. 文档总数
2. 文档大小
3. 要求的索引吞吐量
4. 查询复杂度
5. 预期增长

### 文档总数

由于底层 Lucene 的限制，Solr 中限制了每个索引中包含的文档数目不能超过  $2^{31}$  个。在实际应用中，大多数的 Solr 应用远未达到该上限值之前便遇到了其他问题。随着索引越来越大，在执行排序、分面和缓存过滤器查询等操作时，应用程序的性能和内存便会出现问题。

下面假设一个场景来帮助读者理解为什么对一个大的索引进行分片是有益的。假设一个查询语句能够匹配 1000 万份文档，并且需要按照时间递减的顺序（`sort=timestamp_tdt_desc`）返回文档。当为该查询语句生成结果时，Solr 不得不对 1000 万份匹配查询的文档进行排序，产生包含 10 条记录的一页结果；仅仅对搜索结果做排序就需要大概 80MB 的内存。但是，如果这 1000 万份文档分布在 10 个分片中，每个分片仅需要并行地存储大约 100 万份文档。尽管每个分片都需要重新计算排名前十的结果显得有点重复，但很明显，并行地在 10 个节点的每个节点中对 100 万份文档进行排序，要比在单个节点中顺序地对 1000 万份文档进行排序快得多。当然，这只是一个极端的例子，很少有查询语句能够一次匹配 1000 万份文档，但是上述逻辑也适用于较小的结果集。简单来说，索引中的文档数目越大，分片带来的好处也就越明显。

分片还能帮助你更有效地管理内存。第 5 章介绍了 MMapDirectory，因为它从操作系统的文件系统缓存中读取索引的主要数据结构，所以它是在 64 位的 Linux

和 Windows 操作系统上推荐的文件夹实现类。执行上述操作的基本前提是索引已经被加载到内存中。索引越小，可以被加载到文件系统缓存中的索引内容越多。这个示例也说明了横向扩展要比纵向扩展好。

例如，Amazon EC2 的 3 ml.xlarge 服务器能够提供 12 核的 CPU、45GB 的 RAM 以及良好的网络性能，其定价大概为每小时 1.44 美元。另外，一个具有 8 核 CPU、良好的网络性能以及 64GB 的 RAM (m2.4xlarge) 的系统定价为每小时 1.64 美元。所以将一个索引分布到三个节点中，每个节点大概有 15GB 的内存和 4 核 CPU，这比在一台拥有更大的 RAM 的机器上运行单个索引更加节省费用，同时还提高了性能。从上述示例可以看出，现阶段在云环境中扩展内存总量比使用更多一般的硬件来扩展更加昂贵。当然，这种情况在将来肯定会改变，但是原则是不变的。

但是，如果扩展的是物理服务器（而不是云服务器），则增加额外的 RAM 时可能会得到与上例不同的费用 / 收益曲线，因为相对于添加新的服务器来说，添加物理内存的费用要小得多。随着添加的文档不断增多，最终不得不对应用系统进行扩展。为了提高系统整体的性能 / 费用比，即使是在横向扩展的 Solr 集群中，最佳方案仍然是优化每台服务器。

### 文档大小

在对服务器进行优化时，需要从文档的字段数目和每个字段包含的内容总量的角度来考虑文档的大小。一般情况下，如果文档的字段很多，因为支持搜索的数据结构会消耗很多内存，所以需要设置更多的索引分片。

假设一个搜索应用程序支持多种不同的过滤器。每个缓存过滤器需要 maxDoc/8 字节的空间，那么在一个包含 1.6 亿份文档的索引中，每个过滤器大概需要 20MB 的 RAM 缓存空间。这看起来并没有什么值得忧虑的，但是当你有数百个过滤器时，情况便大大不同了。如果将索引分成 4 个分片，每个分片仅仅包含 4000 万份文档，则每个过滤器仅需要 5MB 的 RAM 空间，而且还将创建过滤器的计算分布到了 4 个节点中并发进行。文档越大，单个分片中存储的文档越少，这样更容易保持合理的资源利用率和系统性能。

### 必要的索引吞吐量

如果需要频繁地添加和更新文档，那么更多的分片有利于均衡索引的负载。假设每秒索引 5000 份文档，如果将索引分成 10 个分片，就相当于 10 台服务器（而不是一台服务器）并发地执行索引操作。因此，一个分片集合每秒索引的文档要比一个非分片集合多，尤其是执行批量更新的时候。但是，分布式索引也有一些开销，所以索引吞吐量可能不会完美地呈线性增长。

### 查询复杂度

如果将索引分为 10 个分片，就可以利用分布式搜索在 10 个分片上并行处理大部分用户请求（如搜索、分面和高亮等）。一方面，随着查询时间变长，且查询复杂度增大，将集合中的文档分成更多的分片并行处理查询请求能够缩短系统的整体响应时间。另一方面，分布式模式下，分组和连接等操作需要特殊的索引分片策略，以确保相关文档都位于相同的分片或服务服务器上。

### 预期增长

索引在未来的几个月会增长多少，这也需要考虑，在初始化时便要规划好索引的空间大小。例如，如果知道索引将在接下来的三个月中增长一倍，则应该根据三个月后的索引数量决定理想的分片数。

但是，并没有一个统一的公式可以方便地计算出应用程序的分片数量。如何权衡上文中介绍的众多参数完全取决于读者，在实际应用中，大部分 Solr 应用都倾向于轻微地过度分片，以适应后期的增长。如果觉得目前搜索应用只需要 4 个分片，不妨考虑设置 6 个，这样可以更好地适应未来的需要。

### 在开发时进行测试

很明显，合理的分片数需要根据实际数据反复试验才能最终决定。但是我们并不希望在生产环境中反复测试不同分片大小的临界值。我们之前讨论过一种方式，即先在开发环境中搭建一个小的 Solr 集群，如一个仅包含两个分片的集群，然后再不断增加分片数，直到同生产环境相符。例如，生产环境中有 2 亿份文档，分布在 10 个分片中，则可以在开发环境中搭建一个包含两个分片的类似的集群，共包含 4000 万份文档（两个环境中的每个分片都大概包含 2000 万份文档）。

你可以按照上述方法一步步地增加开发环境中的分片数目，直到开始出现内存不足、索引速度慢或查询性能差等问题时为止。这样你就可以了解到数据分片的临界点，同时又不必在开发环境中构建一个包含 2 亿份文档的索引进行测试。

## 12.5.2 复制策略

一旦决定了内容的分片数目，就可以如第 3 章（3.4.2 小节）中展示的那样对所有分片进行分布式搜索了。到目前为止，读者接触了每个索引分片上单个副本上的搜索。但是，基于 Solr 集群的性能特征（如系统配置、文档和字段数目、文档大小、查询和文本分析的复杂度等），Solr 集群可能无法满足应用程序查询处理的需求。

### 一个简单的复制情景

如果 Solr 集群每秒仅能处理 100 条查询，但是应用程序要求每秒处理 150 条查



询, 那么问题便出现了。一种解决办法是为索引创建多份副本, 在不同的副本之间进行负载均衡。举一个简单的例子, Solr 服务器有一个每秒能处理 100 条查询的内核, 如果需要每秒处理 150 条查询, 则可以创建一份原始分片的副本, 并在原始分片和副本之间均衡流量。图 12.4 展示了该种配置下服务器的工作方式。

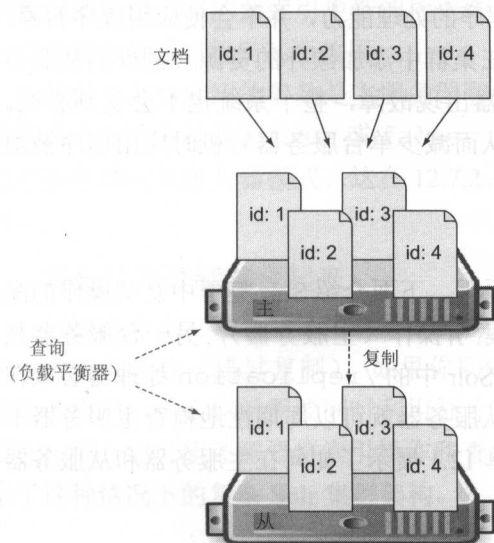


图 12.4 Solr 主内核服务器上的索引复制一份到 Solr 从内核服务器上。两个服务器上都拥有全部的文档。索引副本用于冗余备份或提升查询能力

在图 12.4 中, 所有文档都被发送到同一台服务器 (顶层服务器, 称为主服务器, Master) 中进行索引。上例中的顶层服务器就是指每秒能够处理 100 条查询的原服务器。在图 12.4 中, 原服务器生成的索引从主服务器上被复制到一台新的服务器上, 这台新服务器称为从服务器 (Slave)。因为两台服务器都有相同的索引副本, 所以可以将用户查询的一半发送到主服务器, 另一半发送到从服务器, 这样共同协作后每秒至少能处理 200 条查询, 便能更好地满足应用程序的搜索需求了。

### 将索引与搜索分离

因为从服务器不需要执行消耗资源的索引操作, 所以从服务器每秒钟可以比主服务器处理更多的查询。在许多情况下, 我们会让主服务器专注于创建索引并获得最大索引吞吐量, 同时拥有多个从服务器复制主服务器的索引, 专注于处理搜索请求。在集群中做这样的分离可以使得搜索性能不受文档内容重新索引的影响 (相对来说), 即使重新索引的工作超过了主服务器的负载。当需要在集群中将索引和搜索分离到不同的服务器中, 或当需要增加每秒处理的查询数量时, 可以采取这种主从复制模式。



## 自动容错

利用主从复制模式增加一台服务器并且将索引复制到新服务器上，确实能够提高应用程序的整体查询处理能力，但是如果其中一台服务器宕掉了呢？如果应用程序只有一台服务器，那么很明显应用程序也会暂停。但是如果有多台冗余的服务器，则一台服务器的宕机只会相应地降低应用程序的处理能力，并不会使应用程序暂停。如果想要系统拥有强大的自动容错特性，在集群中添加额外的资源（额外的从服务器）是个不错的主意，这样即使单台服务器出现故障，整个系统也不会受到影响。通过复制，可以在系统中添加这种冗余，从而减少单台服务器宕机时应用程序被迫下线的概率。

## 配置复制

读者应该已经理解了进行复制操作的原因，下面介绍 Solr 集群中复制操作的配置。基本来说，复制要求一台服务器执行索引操作（主服务器），另一台服务器从主服务器上获取索引的副本（从服务器）。Solr 中的 `/replication` 处理器必须在主服务器和从服务器中都进行配置，这样从服务器就可以周期性地检查主服务器上的索引更新，并更新自己的索引。代码清单 12.1 展示了如何在主服务器和从服务器上配置复制。

### 代码清单 12.1 在主从服务器之间建立副本

主服务器的 solrconfig.xml

```
(http://masterserver:8983/solr/core1)
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="enable">true</str>
    <str name="replicateAfter">commit</str>
    <str name="replicateAfter">optimize</str>
    <str name="replicateAfter">startup</str>
  </lst>
</requestHandler>
```

从服务器的 solrconfig.xml

```
(http://slaveserver:8983/solr/core1)
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="slave">
    <str name="enable">true</str>
    <str name="masterUrl">
      http://masterserver:8983/solr/core1/replication
    </str>
    <str name="pollInterval">00:00:15</str>
  </lst>
</requestHandler>
```

主服务器配置中含有不同的 `replicateAfter` 参数，用以表明从服务器应该在何时复制索引，例如，设置为 `commit`、`optimize` 和 `startup`，可以确保从服务器在这些重要操作发生后能够及时地更新索引。在从服务器的配置中，`masterUrl` 参数定义了要复制的及其主服务器所在的 Solr 内核，`pollInterval` 参数定义了对主服务器索引更新的检查频率，可以根据主服务器索引更新的频率将该值设置为数秒或数天甚至数周。

最后一个要介绍的参数是主服务器和从服务器配置中都有的 `enable` 选项，通过将该选项的值设置为 `true` 或 `false`，可以控制副本是否可被用于给定 Solr 内核的主服务器或从服务器模式。这在 12.7.2 小节中介绍 Solr 配置文件的一般化时会用到。

### 将分片和复制结合起来

到目前为止，读者已经了解了如何扩展 Solr 来处理更多的内容（通过分片）和更大的查询负载（通过复制）。如果你同时拥有庞大的数据集和大量想要查询该数据集的用户，那么你就需要同时利用分片和复制来配置 Solr 集群。如果索引的量也很大，那么你可能还需要将索引操作和查询操作分离到不同的服务器上。图 12.5 展示了这种情况下的复杂 Solr 集群架构。

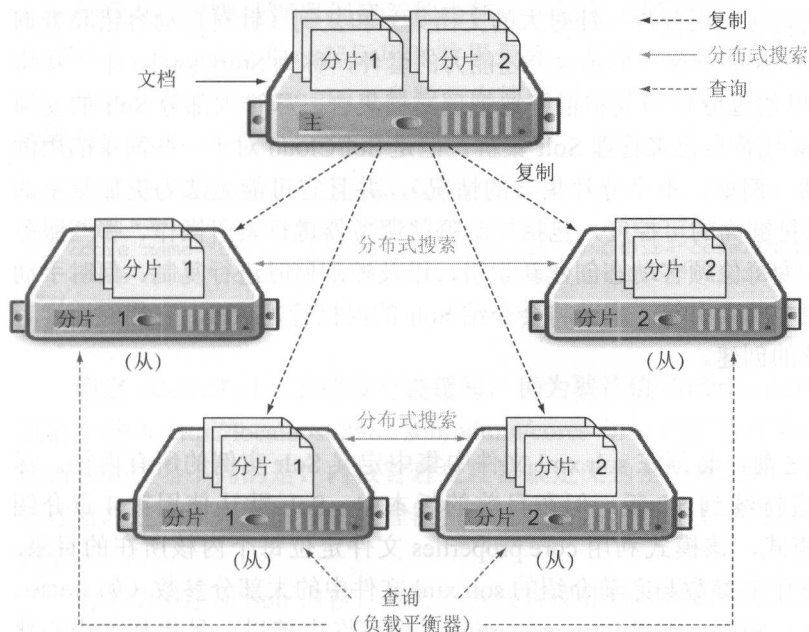


图 12.5 一个复杂的 Solr 集群，包括一台专用的索引主服务器和多个分片形成的两组副本（从服务器）

从图 12.5 可以看出, 配置一个同时处理分片和复制的 Solr 集群, 会为系统维护带来很大难度。因为要在多个手动定义的 Solr 内核之间进行查询负载均衡, 并且确保从服务器的每个 Solr 内核都配置和开启了复制, 所以主服务器上的 Solr 内核将变得十分复杂。如果集群的某个节点出现故障, 会导致集群中的多个其他节点也出现故障。例如, 如果图 12.5 中唯一的主服务器出现故障, 整个集群都会停止更新索引。类似地, 如果一台从服务器出现故障, 任何依赖于该服务器执行分布式搜索的其他从服务器也会无法处理查询请求。

幸运的是, Solr 中 SolrCloud 可以为用户管理这些复杂的事务。本章主要介绍一些手动扩展 Solr 的方法, 用以解释 Solr 中进行扩展的基本原理; 并提供一些工具, 用以处理一些 SolrCloud 目前还不能处理的用例。第 13 章将会介绍如何利用 SolrCloud 代理系统的自动容错及分布式查询路由, 来使得 Solr 的扩展更加可控。无论采取哪种扩展方法, 都应当了解如何在不重启 Solr 的情况下与 Solr 内核进行交互。下一节将详细介绍 Solr 的内核管理 API, 该 API 为管理 Solr 内核提供了丰富的特性。

## 12.6 Solr内核管理

上一节介绍了如何利用分片(针对大的文档集)和复制(针对自动容错和查询负载)来扩展 Solr。Solr 中有一个包含上述能力的套件, 称为 SolrCloud (下一章将深入介绍), 它可以通过分片和复制很好地管理文档集合。尽管大部分 Solr 的使用者都想让 SolrCloud 代替自己来管理 Solr 集群, 但是 SolrCloud 对于一些简单的用例来说可能过于复杂(例如, 单个分片集合的情况), 并且它可能无法为更加复杂的 Solr 实现需求提供足够高的可控度, 包括控制哪些服务器进行索引操作、哪些服务器进行搜索操作, 为每位顾客动态创建新索引, 以及控制何时进行复制、何时手动复制外部数据资源中的索引等。本节将会介绍 Solr 的内核管理 API, 它能够让读者手动控制 Solr 内核的创建。

### 定义内核

在 Solr 4.4 版之前, 必须在 solr.xml 文件中集中定义 Solr 实例的所有内核。尽管该项特性将一直持续到 5.0 版, 但在目前的版本中, Solr 默认使用第 4 章介绍的内核自动发现模式, 该模式利用 core.properties 文件定位每个内核所在的目录。core.properties 文件中的参数与之前介绍的 solr.xml 文件中的大部分参数(如 name、config 和 schema)相同, 但是 core.properties 文件允许内核以一种更加去中心化的方式被定义, 而不是被迫都定义在 solr.xml 文件中。

### 通过内核自动发现模式管理内核的局限性

尽管新的 `core.properties` 文件的内核自动发现模式更加适应未来的需求，但是目前这种方式仍有几点局限性。最主要的一个局限便是不能共享配置文件（在 Solr 4.7 中仍然是这样）。因为这一局限，本节的示例假定读者都使用 `solr.xml` 文件来配置内核，这样可以定义一个 `instanceDir` 参数，将该参数设为相同路径便可以实现配置文件的共享。例如，内核创建操作不允许在不使用 `solr.xml` 文件配置内核的情况下根据共享的配置文件动态创建新的内核。这些内核自动发现模式的局限将在 Solr 5.0 版发布之前得到解决。

也就是说，`solr.xml` 文件中的某些功能在新的 `core.properties` 文件中还不被支持。尽管本书附带所有的示例 Solr 内核配置文件都使用新的内核自动发现模式，但是内核管理处理器（后面将介绍）仍然是与 `solr.xml` 文件契合度最高的，尤其是创建新的内核时（详见上面的“通过内核自动发现模式管理内核的局限性”）。因此，请注意，本节中的所有示例都假定了读者使用的是 `solr.xml` 文件来配置内核。

### 利用内核管理 API 创建内核

除了预先在 `solr.xml` 文件或 `core.properties` 文件中定义 Solr 内核，也可以要求 Solr 利用内核管理处理器创建新的内核。正如上文提到的，在 Solr 4.7 中内核管理处理器与 `solr.xml` 文件一起运用时效果最好，并且 `solr.xml` 文件允许定义一个多内核之间共享的 `instanceDir` 参数。内核管理处理器的路径可以在 `solr.xml` 文件中被覆写。

```
<solr persistent="true" sharedLib="lib">
  <cores adminPath="/admin/cores">
    <core name="core1" instanceDir="shared" dataDir="core1/data" />
    <core name="core2" instanceDir="shared" dataDir="core2/data" />
  </cores>
</solr>
```

假定 `adminPath` 的值没有被覆写，仍为默认值 `/admin/cores`，则可以在浏览器中输入 `http://localhost:8983/solr/admin/cores` 访问内核管理处理器的页面。与其他请求处理器不同的是，内核管理处理器被定义为全局处理器，而不是单一的内核处理器。这是因为内核管理处理器负责管理内核，因此不能运行在任何单个内核的内部。

为了动态添加 Solr 内核，需要向内核管理处理器发送 `CREATE` 请求。

```
http://localhost:8983/solr/admin/cores?
  action=CREATE&
  name=coreX&
  instanceDir=path_to_instance_directory&
  config=solrconfig_file_name.xml&
  schema=schema_file_name.xml&
  dataDir=data&
  loadOnStartup=true&
  transient=false
```

对于 CREATE 请求，只需要定义内核名称和 instanceDir 参数。如果需要引用另一个文件夹中的 schema.xml 文件或 solrconfig.xml 文件，可以设置 schema 和 dataDir 参数。另外，Solr 支持延迟加载（和卸载）Solr 内核，可以通过指定 loadOnStartup=true|false 来开启或关闭延迟加载，并且如果系统需要释放资源的话，可以通过指定 transient=true|false 来开启或关闭内核的卸载。

请注意，如果在 solr.xml 文件中设置了 persist=false，那么通过内核管理处理器创建的所有 Solr 内核在 Solr 重启之后就都不存在了，这可能会让你很失望，所以必须确定是否需要在动态地创建内核的同时保存配置。另外需要注意的是，一个 Solr 内核只能被创建一次，如果两次或多次用相同的参数发起 CREATE 请求，Solr 就会抛出异常。

除了创建 Solr 内核，还可以在已有的 Solr 内核上执行重载、重命名、名称置换、卸载、索引切分和索引合并等操作。

### 重载内核

如果要更改 Solr 内核的配置文件（如 solrconfig.xml 文件、schema.xml 文件以及其他 conf/ 文件夹下的文件），这些更改在 Solr 重启之前不会立即生效。但是，发起 RELOAD 请求，会让 Solr 内核重新加载配置文件和初始化：

```
http://localhost:8983/solr/admin/cores?
  action=RELOAD&
  core=coreX
```

在重载过程中，当前版本的 Solr 内核将会保持活跃并接受用户请求，直到新加载的内核可以使用，这样在 Solr 内核的切换过程中对用户请求的处理就不会中断。但是，有一些设置的更改只有在重启 Solr 之后才会被应用，例如 solrconfig.xml 文件中设置 Solr 内核路径的 dataDir 参数和其他一些 IndexWriter 的设置，其他大部分设置的更改都可以通过 RELOAD 方式得到更新。另外，在 Solr 内核的重载过程中，所有缓存和请求处理器的统计数据都将丢失。

### 重命名和置换内核

有时候，重命名一个 Solr 内核是很有帮助的。例如，你有一份老版本的索引，

需要注明它是过时的，可以利用 RENAME 来操作：

```
http://localhost:8983/solr/admin/cores?
  action=RENAME&
  core=coreX&
  other=coreX_old
```

在该示例中，名为 coreX 的内核被重命名为 coreX\_old。如果要载入新的名为 coreX\_new 的 Solr 内核来取代旧的 Solr 内核，可以将该过程拆解为两次重命名操作：第一次将旧内核重命名为其他名字（从 coreX 改为 coreX\_old），第二次将新内核重命名为旧内核之前的名字（从 coreX\_new 改为 coreX）。这种方式最明显的问题就是在两次重命名过程中，没有内核叫 coreX，意味着此时任何发往 coreX 的请求都将失败。

为了解决这个问题，可以使用 Solr 中的 SWAP 操作，它可以通过交换两个 Solr 内核的名字来保证重命名操作的原子性。在之前的示例中，需要执行两次重命名操作。首先，需要发起 SWAP 请求将 coreX\_new 和 coreX 相互交换：

```
http://localhost:8983/solr/admin/cores?
  action=SWAP&
  core=coreX&
  other=coreX_new
```

一旦交换了两个 Solr 内核，就可以监控新的 Solr 内核（现在被称为 coreX）是否按预期运行。如果发现问题，可以再次交换两个内核。因为可以在新内核中重新索引整个数据集并进行实时置换，所以如果出现任何问题，都可以在极短的时间内置换回去，这种方式降低了对 Solr 做大规模变更的危险性（如重新索引数据集）。

假定新的 coreX 如预期运行，可以使用 SWAP 将之前的 Solr 内核（目前被不正确地命名为 coreX\_new）重命名为一个更贴切的名字，如 coreX\_old，或者如果不再需要的话可以直接删掉它。

### 卸载和删除内核

一旦决定了不再需要加载某个 Solr 内核，可以向内核管理处理器发送 UNLOAD 请求：

```
http://localhost:8983/solr/admin/cores?
  action=UNLOAD&
  core=coreX_old&
  deleteInstanceDir=false&
  deleteDataDir=false&
  deleteIndex=false
```

在 UNLOAD 请求中只有 core 参数是必需的。默认地，卸载一个 Solr 内核并不会删除相关的索引、数据文件夹和实例文件夹。这意味着可以轻易地卸载一个内核，



并且, 如果 `instanceDir` 的值相同 (如果覆写了 `schema` 和 `config` 参数, 则 `schema` 和 `config` 参数也要相同), 就可以利用 `CREATE` 重新加载该内核。如果要完全删除掉与 Solr 内核相关的数据和配置, 可以将 `deleteInstanceDir=true` 参数添加到 `UNLOAD` 请求中。如果只是想删除数据文件夹, 可以添加 `deleteDataDir=true`, 如果只想删除数据文件夹中的索引 (但不删除整个数据文件夹), 可以指定 `deleteIndex=true`。

### 切分和合并索引

有时, 当前 Solr 内核中的数据可能会不断增长, 甚至需要将它们切分到多个分片中。我们可以按照常规方式先设置新的分片, 再重新索引数据, 不过 Solr 提供了将 Solr 内核切分为多个分片的机制, 并且不需要重新索引数据。例如, 要将一个内核切分为三个内核, 可以利用 `SPLIT` 进行操作:

```
http://localhost:8983/solr/admin/cores?
  action=SPLIT&
  core=oldCore&
  targetCore=newCore1&
  targetCore=newCore2&
  targetCore=newCore3
```

被切分的 Solr 内核的名称应当被传递给 `core` 参数, 并且必须要根据分片数指定至少一个 `targetCore` 参数, 每个 `targetCore` 参数表明一个分片。需要注意的是, 要想上述命令有效的话, 必须保证每个 `targetCore` 指定的目标内核都已经被创建, 并且当前在 Solr 服务器中处于活跃状态 (可以先执行 `CREATE` 操作)。如果所指定的目标内核中的某个内核已经有了文档, 则这些文档会与被切分的新文档合并。因为可以将一个索引切分至单个 `targetCore`, 所以也可以使用这个命令创建当前内核的备份, 存储到一个目标内核中。

如果目标内核都还没有运行, 则可以采取另一种方式, 即在请求命令中指定分片在磁盘存储的目标地址:

```
http://localhost:8983/solr/admin/cores?
  action=SPLIT&
  core=oldCore&
  path=/path/to/newCore1/data/&
  path=path/to/newCore2/data/&
  path=path/to/newCore3/data/
```

如果采取第二种方式, 可以在 `SPLIT` 操作结束后向内核管理处理器提交一个 `CREATE` 操作, 以激活新的 Solr 内核。需要注意的是, 这种方式 (通过指定路径切分索引) 假定指定路径下的索引已经被卸载了, 并且不接受任何更新。否则, 如果指定路径下的索引未被卸载且接受更新, 则会导致索引冲突。如果目标内核已经被



加载了, 则应该尽量使用 `targetCore` 参数来切分索引, 确保所有的索引的写操作都是同步的, 并且不会发生索引冲突。

与切分索引相对应, 如果需要合并索引, Solr 的内核管理 API 提供了 `Merge-Indexes` 操作:

```
http://localhost:8983/solr/admin/cores?
  action=MERGEINDEXES&
  core=newCore&
  srcCore=oldCore1&
  srcCore=oldCore2&
  srcCore=oldCore3
```

当使用 `MERGEINDEXES` 操作时, 必须指定存储合并后的索引的新内核 (`core` 参数), 以及一个或多个执行合并操作的内核 (`srcCore` 参数)。

至于 `MERGE` 操作, 也可以在索引未载入 Solr 中的前提下合并。要实现这点, 需要将 `srcCore` 参数的值替换为指向磁盘上索引路径的参数 `indexDir`。

```
http://localhost:8983/solr/admin/cores?
  action=MERGEINDEXES&
  core=newCore&
  indexDir=/path/to/oldCore1/data/&
  indexDir=path/to/oldCore2/data/&
  indexDir=path/to/oldCore3/data/
```

关于切分索引, 很重要的一点是在切分索引时, 更新不会被写入 `indexDir` 文件夹下的索引文件中。如果要合并的内核已经被加载了, 并且正在 Solr 实例中运行, 那么最好使用 `srcCode` 参数, 而不要使用 `indexDir` 参数, 以确保不会出现索引冲突。另外, 执行带有一个 `targetCore` 参数的 `SPLIT` 操作或者带有一个 `core` 参数的 `MERGEINDEXES` 操作, 将会产生相同的结果: 一个原内核的副本将同目的内核中已有的索引文档合并。

### 获取内核状态信息

除了利用上述方法操纵 Solr 内核, 还可以通过 `STATUS` 获取一系列 Solr 内核的状态:

```
http://localhost:8983/solr/admin/cores?
  action=STATUS
```

执行上述命令, Solr 将返回服务器上活跃的 Solr 内核列表, 以及它们的配置信息、索引大小和常用性能信息。如果想要了解某个 Solr 内核的状态信息, 可以添加 `core=coreName` 参数来限制仅返回指定内核的状态信息。

获取 Solr 内核的状态信息, 可以帮助 Solr 使用者在操作 Solr 内核之前更好地了解每个 Solr 内核的状态。任何连接了服务器的监控工具都可以使用该方法获取相

关信息。如果需要管理的服务器数量比较少,则可以手动管理和配置 Solr 内核;如果服务器数量较多,则可以参考下一节中介绍的一些技巧,这些技巧有助于降低将 Solr 集群扩展到多台服务器的复杂度。

## 12.7 管理服务器集群

当 Solr 实例多于一个时,管理 Solr 集群就会变得更加复杂。在分布式的搜索环境中,如果单台服务器宕机,那么其他需要利用该服务器进行分布式搜索的服务器上的搜索操作也会失败。同样的,如果一个或多个副本有了影响应用程序的问题,就需要一些方法来确定哪些服务器是健康的,并将其留作生产环境的负载均衡器。此外,当开始增加更多的服务器时,对每台服务器的独特配置(solrconfig.xml、schema.xml 文件等)进行管理是非常复杂的,这将成为麻烦的维护问题。本节将讨论对这些问题的解决有所帮助的技巧。

### 12.7.1 负载均衡器和 Solr 健康检查

假设你有一个包含分片和副本的 Solr 分布式集群,你想要采取某种方式均衡集群中各服务器之间的流量。如果所在公司有一个硬件负载均衡器,则可以为每一台 Solr 服务器设置一个虚拟 IP 地址。如果你使用 Amazon EC2 的云端服务,则可以使用 Amazon 的 Elastic Load Balancer 以相似的方式配置负载均衡。如果足够幸运正在执行一个比较小的操作,则可以考虑安装 HAProxy 作为负载均衡器。无论采取哪种解决方案,都会需要某种“健康检查”选项——负载均衡器来查看服务器,判断服务器是处于健康状态,应该继续提供服务,还是处于故障状态,应当被移除。

为了方便这类健康检查,Solr 维持了一个 Ping 请求处理器,它可以通过在 solrconfig.xml 文件中添加特殊的 requestHandler 选项进行启动。Ping 请求处理器负责决定服务器是否能够接收流量,并成功地执行查询。要启用 Ping 请求处理器,需要确保已经在 solrconfig.xml 文件中对它进行了配置。

```
<requestHandler name="/admin/ping" class="solr.PingRequestHandler">
  <lst name="invariants">
    <str name="q">choose_any_query</str>
    <str name="shards">
      localhost:8983/solr/core1,remotehost:8983/solr/core2
    </lst>
    <str name="healthcheckFile">server-enabled</str>
  </requestHandler>
```

启用 Ping 请求处理器之后,现在所要做的就是将负载均衡器指向每台服务器的 `http://servername:8983/solr/admin/ping` 地址,如果负载均衡器收到 Status 200 OK

响应，则意味着 Solr 集群处于健康状态；如果收到 HTTP 错误代码（200 之外的任何状态代码），则意味着某一服务器出现了问题，负载均衡器会停止向该服务器发送流量。

向负载均衡器中添加或移除服务器（当 Solr 处于健康状态时）是一个十分有用的可选功能，利用 `healthcheckFile` 设置可以开启该功能。如果此设置被包含在请求处理器的配置中，那么执行健康检查查询之前，Ping 请求处理器会先检查磁盘中是否存在指定名字的健康检查文件（该文件位于 Solr 内核的 `conf/` 文件夹中）。如果该文件不存在，Ping 请求处理器将会自动返回错误代码，且不会执行该查询。

利用这个可选的健康检查文件的特性，我们可以在不关闭 Solr 的前提下，从负载均衡器中精确地移除某台 Solr 服务器。你可以手动创建和删除 Solr 服务器上的文件，或者也可以请求 Ping 请求处理器来完成。

- 开启 ping : `http://servername:8983/solr/core1/ping?action=enable`。
- 关闭 ping : `http://servername:8983/solr/core1/ping?action=disable`。

然后，可以向 Ping 请求处理器发起请求，以获取健康检查文件的状态，或查看服务器是否健康。

- 检查健康检查文件 : `http://servername:8983/solr/core1/ping?action=status`。
- 检查健康检查文件（如果已经开启）和查询执行是否成功 : `http://servername:8983/solr/core1/ping`。

读者可能已经注意到，我们在 Ping 请求处理器的配置中加入了一个 `shards` 参数。Ping 请求处理器支持分布式搜索，这意味着如果任何定义或通过使用（可选的）`shards` 参数定义的分片没有成功响应，那么健康检查便会失败。如果所有的服务器都引用了相同的故障分片，则会导致整个集群几乎在同一时间失去响应，因为它们都无法成功地执行分布式测试查询。因为需要所有的分片都成功地执行请求，所以这便是大多数情况下我们想要的健康检查效果。

如果正在负载均衡器的后面运行 Solr，Ping 请求处理器可以帮助避免与负载均衡器的直接交互（这将超越最初设立的健康检查），从而使得与负载均衡器的交互变得简单。在 Solr 中添加或卸载内容都可以通过一个简单的 API 来控制。随着 Solr 的不断扩展，为不同的 Solr 服务器部署许多不同的配置文件并进行版本管理会成为 Solr 维护的一大难题。下一小节将对如何管理这种复杂性给出一些小建议。

## 12.7.2 通用配置 vs. 自定义配置

Solr 有很强的自定义性。每个 Solr 内核都可以设置独立的 `schema.xml` 文件和 `solrconfig.xml` 文件，而且硬编码到 `schema.xml` 文件中，每个字段在每个独立的用例中都有相应的名称和属性。这对于小规模 Solr 部署而言很合适。一旦开始添加成

百上千台的服务器，一份很大的自定义的 Solr 配置文件便会变得难以管理。每次需要在具有不同配置的众多服务器上升级或者重新部署某个版本的 Solr 时，你都不得不选择手动更新每个 Solr 的特定配置文件（core.properties 文件、solrconfig.xml 文件、schema.xml 文件等）。

降低维护大量 Solr 服务器所需的相关操作成本的一个策略是尽可能地将配置文件通用化。例如，公司在基于 Solr 构建云端搜索服务时往往会将所有可用的字段定义为动态字段，允许顾客随时创建新的字段，同时不必修改任何已经部署的 Solr 实例上的 Solr 模式。这样，系统中的每台 Solr 服务器都可以拥有完全相同的 schema.xml 文件，也意味着将 schema.xml 文件重新部署到任意一个集群就相当于部署到了所有集群。

但是，这种通用配置也存在一些问题，最大的问题是需要预先定义所有字段类型为动态字段。这需要假设有一个文本字段，需要创建一些索引和存储了该字段并包含规范的变量（与 omitNorms="true" 相反）。如果想要支持从客户端应用这些属性的任意组合，就需要为这个字段类型在 schema.xml 文件中添加以下内容：

```
<dynamicField name="*_t" type="text"
  indexed="false" stored="false" omitNorms="true" />
<dynamicField name="*_t_i" type="text"
  indexed="true" stored="false" omitNorms="true" />
<dynamicField name="*_t_s" type="text"
  indexed="false" stored="true" omitNorms="true" />
<dynamicField name="*_t_n" type="text"
  indexed="false" stored="false" omitNorms="true" />
<dynamicField name="*_t_is" type="text"
  indexed="true" stored="true" omitNorms="true" />
<dynamicField name="*_t_in" type="text"
  indexed="true" stored="false" omitNorms="false" />
<dynamicField name="*_t_sn" type="text"
  indexed="false" stored="true" omitNorms="true" />
<dynamicField name="*_t_isn" type="text"
  indexed="true" stored="true" omitNorms="false" />
```

尽管这样通用的配置使得 Solr 的部署脚本更加简单，但同时也要求我们在性能和简单之间做权衡。是否需要支持每种字段类型的所有可能配置选项？例如，对于数值字段类型，支持的数值精度是多少？在自定义的 schema.xml 文件中，可以控制所有的配置细节，但是在一个统一配置文件中，只需要谨慎选择所提供的自定义级别，同时需要避免之前示例中出现的不兼容选项。

除了将模式通用化，还可以将 solrconfig.xml 文件通用化。因为不同的服务器会有不同的资源需要，也会有不同的设置，所以 solrconfig.xml 文件的统一比 schema.xml 文件的统一更加具有挑战性，但是仍然可以通过细心使用特定内核的属性来实现。

本章在代码清单 12.1 中介绍了如何在主服务器和从服务器之间设置复制，在该代码清单中，必须要向从服务器上的 `/replication` 处理器指定被复制的服务器。在大型服务器集群中，可能有从不同的主服务器复制过来的从服务器。代码清单 12.2 展示了相关的配置：

代码清单 12.2 通过使用变量生成 `solrconfig.xml`

通用 `solrconfig.xml`

```
...
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="enable">${master.replication.enabled:false}</str>
    <str name="replicateAfter">commit</str>
    <str name="replicateAfter">optimize</str>
    <str name="replicateAfter">startup</str>
  </lst>
  <lst name="slave">
    <str name="enable">${slave.replication.enabled:false}</str>
    <str name="masterUrl">
      http://${masterserver:}/solr/${solr.core.name}/replication
    </str>
    <str name="pollInterval">00:00:15</str>
  </lst>
</requestHandler>
...
```

从服务器的 `solrcore.properties` 文件

```
...
slave.replication.enabled="true"
masterserver="masterserver:31000"
...
```

主服务器的 `solrcore.properties` 文件

```
...
master.replication.enabled="true"
...
```

可以看出，针对特定从服务器的 Solr 内核的属性已经从 `solrconfig.xml` 文件中被抽象成了变量，定义在 Solr 内核的 `solrcore.properties` 文件中。如果在 Solr 内核的 `conf/` 文件夹中存在 `solrcore.properties` 文件，则该文件将被用于传递特定 Solr 内核的属性。这意味着同一份 `solrconfig.xml` 文件可以被用于整个 Solr 体系的任意服务器上。事实上，我们还应该注意到在 `solrconfig.xml` 文件中定义了 `${master.replication.enabled:false}` 参数和 `${slave.replication.enabled:false}` 参数。这种语法形式表示，除非在其他文件中（在本例中，为 `solrcore.properties` 文件）明确地定义了这两个参数的值为 `true`，否则两个参数的默认值都是 `false`（禁用特定主服务器或从服务器的复制）。因此，无论是主服务器

还是从服务器，也无论从服务器指定的是哪个主服务器，Solr 体系中的任何服务器都能够共享同一份 `solrconfig.xml` 文件。这极大地减少了需要维护的配置文件的数量，使得在多台服务器之间部署 Solr，以及在不同版本之间升级配置文件都变得更加容易。

一旦配置好了服务器，就可以准备开始在应用程序中使用 Solr 了。下一节将介绍向 Solr 发送请求的加速方式。

## 12.8 Solr的查询与交互

到目前为止，本书中的示例主要是通过 REST API 与 Solr 交互，但是还有很多更好的交互方法。本节将会介绍多种通过 Web 应用程序与 Solr 进行交互的方法。

### 12.8.1 REST API

贯穿本书的标准 REST API 是最常见的与 Solr 交互的机制。因为该 API 仅依赖于 HTTP 协议，所以可以利用任何能通过网络连接 Solr 服务器的编程语言与 Solr 交互。

正如第 7 章介绍的（见 7.7.1 小节），可以从 Solr 中以某种已经序列化为许多编程语言（如 Python、Ruby、PHP、Javascript/JSON 等）的格式接收响应数据。尽管有各种有用的响应格式，但如果你是刚开始使用 Solr，就不得不痛苦地利用自己偏好的语言编写客户端代码将查询发送到 Solr 中。幸运的是，早期的开发者已经编写了客户端库来处理与 REST API 的交互，你可在应用程序的代码中直接使用对象。下一小节将介绍一些可用的特定语言的库。

### 12.8.2 可用的 Solr 客户端库

尽管 Solr 的 REST API 暴露了 Solr 的所有可用的功能特性，但还有一些库可以将 REST API 的交互抽象成对象。并非所有的库都能够跟上最新的 Solr 发布版（并且有几个库似乎目前已经不再维护了）。并非所有的库都能跟得上 Solr 的最新版本，其中有些库现在似乎已经不再维护了。但是，Solr 在公共 API 方面保持很好的向后兼容性，也就是说，即便某个库过时了，你依然有可能正常使用它的绝大多数功能。

一些流行的客户端库包括 RSolr（Ruby）、Solarium（PHP）、ScalaLikeSolr（Scala）、SolPython（Python）、SolrJSON（Javascript）、SolrNet（.Net）及 SolrPerl（Perl）等。这些语言包含多个客户端以供选择，我们可以在 <http://wiki.apache.org/solr/IntegratingSolr> 上找到一份公开的列表。



除上述这些语言外, Solr 还自带有一个 Java 客户端库, 称为 SolrJ (在第 5 章中已经介绍过)。由于 SolrJ 的开发语言为 Java, 所以它允许使用二进制对象表示 (默认), 而不要求使用典型的 XML 请求 / 响应格式。SolrJ 将下一小节中详细介绍。

### 12.8.3 使用 SolrJ

正如第 5 章介绍的, SolrJ 允许用户利用自己的应用程序代码与 Solr 通信。Solr 中内置有 SolrJ, 这意味着当你获取 Solr 代码或者下载一个发布版时, 相应版本的 SolrJ 也会被一起下载。SolrJ 也可以使用 (可选) Java 二进制请求 / 响应格式, 减少与其他通信格式相关的序列化和去序列化消耗。使用 SolrJ, 可以通过 HTTP 连接进行通信, 但是也可以将 Solr 内嵌在应用程序中, 这种情况下就不需要暴露服务器上的 HTTP 连接。

#### 将 SolrJ 添加到项目中

SolrJ 被编译成 apache-solr-solrj-\*.jar 文件, 存储于 \$SOLR\_INSTALL/dist/ 文件夹中, 其中 \* 代表 Solr 的版本。SolrJ 包含了对位于 \$SOLR\_INSTALL/dist/solrj-lib/、\$SOLR\_INSTALL/dist/ 及 \$SOLR\_INSTALL/example/lib/ 文件夹中的其他库的依赖。因此需要在 classpath 中添加这些依赖的路径, 或者使用像 Maven 这样的依赖管理器进行自动管理。如果正在使用 Maven, 可以在项目的 pom.xml 文件中添加以下内容 (指定当前 Solr 的版本) 来添加 SolrJ 及其依赖:

```
<dependency>
  <groupId>org.apache.solr</groupId>
  <artifactId>solr-solrj</artifactId>
  <version>4.7.0</version>
</dependency>
```

一旦将 SolrJ 添加到了项目中, 则需要创建一个 SolrServer 对象与 Solr 进行交互。可以创建 HttpSolrServer 或 EmbeddedSolrServer, 至于到底选择哪一个, 取决于你想要连接正在运行的 Solr 实例 (通过 HTTP), 还是想要将 Solr 嵌入到当前系统中 (而不使用 HTTP 连接)。

#### 使用 SolrJ 通过 HTTP 连接 Solr 服务器

SolrJ 最常见的使用方式是通过 HTTP 连接一个已经在运行的 Solr 服务器。在这种方式下, SolrJ 可以像其他的客户端库一样与 Solr 进行交互, 同时又不禁止 SolrJ 以外的客户端发起的其他连接。

要 想 让 SolrJ 知 道 Solr 服 务 器, 需 要 创 建 一 个 SolrServer 对 象。SolrServer 对象有几种不同的实现方式可供选择, 表 12.1 列出了所有类型。



表 12.1 SolrJ 可用的 SolrServer 实现

SolrServer 实现	说明
HttpSolrServer	默认的 SolrServer 用于与 Solr 的端对端请求
ConcurrentUpdateSolrServer	用于发送更新 / 删除请求。按序列进行文档更新的批处理，以提高索引吞吐量
LBHttpSolrServer	启用跨 Solr 端点的请求负载平衡
CloudSolrServer	通过 Solr 端点的 ZooKeeper 动态发现实现跨 SolrCloud 集群的负载平衡
EmbeddedSolrServer	以嵌入模式在当前系统中使用 Solr，无须启用 HTTP 访问

从表 12.1 大概可以看出，HttpSolrServer 是最基本的 SolrServer 的实现方式。我们可以通过 HttpSolrServer 发送所有请求，它们将针对单个指定的 Solr 端点立即执行。在 Solr 索引多份文档时，如果每次只向 Solr 发送一份文档就会效率很低，而 ConcurrentUpdateSolrServer 可以形成文档队列，当队列中的文档达到某个数量或超过一定时间时，这些文档就会被批量地发送给 Solr。因为 ConcurrentUpdateSolrServer 是为向 Solr 发送更新请求而专门设计的，所以对于查询请求，推荐使用其他的 SolrServer 实现。如果你有一个多服务器的 Solr 集群，则可以使用 LBHttpSolrServer 来代替，它会在指定的 Solr 端点之间实现负载均衡。如果你使用的是 SolrCloud（详见第 13 章），那么 CloudSolrServer 将会根据从 ZooKeeper 获得的信息自动决定在哪些服务器之间实现负载均衡。如果要将 Solr 内嵌到应用程序中，而不是作为一个独立的 Web 应用程序运行，则可能需要使用 EmbeddedSolrServer。

通过 SolrJ 与 Solr 交互

从上一小节可以看出，根据应用程序和 Solr 配置的情况，可以选择许多不同的 SolrServer 实现方式。因为 HttpSolrServer 是最常见的选择，所以本节将使用它来展示一些基本的 SolrJ 的操作。在第 5 章（5.5.2 小节）已经介绍过如何使用 SolrJ 向 Solr 发送文档并进行索引。除了发送文档更新，SolrJ 也可以直接利用 Java 调用所有的 Solr 请求。代码清单 12.3 展示了如何执行基本的文档更新、删除和搜索。

代码清单 12.3 通过 SolrJ 与 Solr 进行交互

```
SolrServer server = new
    HttpSolrServer("http://localhost:8983/solr/collection1");
server.deleteByQuery( "*" );
```

实例化一个 SolrServer，与 Solr 内核对话。

删除所有之前就存在的文档。

```

SolrInputDocument doc1 = new SolrInputDocument();
doc1.addField( "id", "1", 1.0f );
doc1.addField( "cat", "health", 1.0f );
doc1.addField( "price", 100 );

SolrInputDocument doc2 = new SolrInputDocument();
doc2.addField( "id", "2", 1.0f );
doc2.addField( "cat", "entertainment", 1.0f );
doc2.addField( "price", 150 );

SolrInputDocument doc3 = new SolrInputDocument();
doc3.addField( "id", "2", 1.0f );
doc3.addField( "cat", "entertainment", 1.0f );
doc3.addField( "price", 99 );

Collection<SolrInputDocument> docs = new ArrayList<SolrInputDocument>();
docs.add( doc1 );
docs.add( doc2 );
docs.add( doc3 );

server.add( docs );
server.commit();

SolrQuery query = new SolrQuery();
query.setQuery( "*:*" );
query.addSortField( "price", SolrQuery.ORDER.desc );

QueryResponse rsp = server.query( query );
SolrDocumentList docs = rsp.getResults();
...

SolrQuery solrQuery = new SolrQuery()
    .setQuery("*:*")
    .setFacet(true)
    .setFacetMinCount(1)
    .setFacetLimit(10)
    .addFacetField("cat");

QueryResponse rsp = server.query(solrQuery);
...

server.deleteByQuery("*:*");

```

创建一个文档，  
添加字段。

三个字段参数：  
name、value 与  
boost。

将文档全部发送  
给 Solr。

提交这些文档，  
以便它们能被  
搜索到。

查询所有  
文档。

执行搜索，处理  
搜索结果。

为查询添加一  
种排序方法。

为该查询启用  
分面过滤。

一旦结束，清除  
所有文档。

向集合添加每个文档  
(批量更新)。

SolrJ 支持大多数的 Solr 操作，代码清单 12.3 仅仅是让读者对如何在 Java 代码中与 SolrJ API 交互有一个基本的认识。想要了解 SolrJ API 提供的其他功能的信息，请阅读 SolrJ 的 JavaDoc 文档。

### 利用 SolrJ 将 Solr 嵌入应用程序中

我们可以利用 SolrJ 与一个正在运行的 Solr 服务器通过 HTTP 交互，除此之外，在 12.2.2 小节中介绍过也可以直接在另一个 Java 应用程序中嵌入 Solr。既然我们已经学习了如何使用 SolrJ，接下来将它嵌入到 Solr 中应该是很自然的，因为在写 SolrJ 代码时，所有的都需要使用 EmbeddedSolrServer 而不是

HttpSolrServer。正如代码清单 12.4 所示，因为可以直接启动 Solr 内核，所以必须将 Solr 的配置文件传递到 EmbeddedSolrServer 中。

代码清单 12.4 将 Solr 内嵌在其他 Java 应用中

```
File home = new File( "/path/to/solr/home" );
File file = new File( home, "solr.xml" );
CoreContainer container = new CoreContainer();
container.load( "/path/to/solr/home", file );
EmbeddedSolrServer server = new EmbeddedSolrServer(
    container, "core1" );

SolrQuery query = new SolrQuery();
query.setQuery( "*" );
...
QueryResponse rsp = server.query( query );
SolrDocumentList docs = rsp.getResults();
```

如你所见，嵌入式的 Solr 利用 SolrJ 库的方式同传统的 Solr 服务器实例基本相同。代码上的唯一区别就是嵌入应用程序中时创建的是 EmbeddedSolrServer，而不是创建 HttpSolrServer，并要传入 Solr 主文件的路径、solr.xml 文件的路径以及要发送请求的 Solr 内核的名字。

嵌入式的 Solr 服务器相对于其他 SolrServer 类型有一个很大的缺陷：它不支持分布式搜索。客观地说，如果将 Solr 嵌入到应用程序中，本身就说明不太可能要在多台服务器之间进行分片，但不支持在同一服务器的多个 Solr 内核之间进行搜索，这确实是一个重大的缺陷。

### SolrJ 版本和序列化

Solr 在兼容 SolrJ 方面做得不错，新版本的 SolrJ 总是能够兼容旧版本的 Solr。这在很大程度上是因为 SolrJ 使用 Solr 的 XML 请求和返回格式与 Solr 交互，并且旧版本支持的功能也被新版本支持。

也就是说，可以将默认的传输格式从 XML 改为 Java 二进制格式，这将会大大地降低与 Solr 请求相关的序列化 / 去序列化的成本。要实现这一点，需要为 HttpSolrServer 设置请求解析器和请求格式器：

```
SolrServer server = new
    HttpSolrServer("http://localhost:8983/solr/collection1");
server.setParser(new BinaryResponseParser());
server.setRequestWriter(new BinaryRequestWriter());
```

如果将请求格式器和响应解析器改为使用 Java 二进制格式，则可能会在使用一个与 Solr 版本不同的 SolrJ 时遇到问题。如果你需要使用不同的版本（例如，因为升级的原因），可能需要坚持使用默认的 XML 请求和响应格式。

## 12.9 监控Solr的性能

当 Solr 被部署到真实流量压力下的生产环境中时，理解 Solr 集群的性能特点就显得很重要。如果在较小的负载下，查询速度仍然太慢，则可能需要进一步切分内容，降低查询的复杂度，或者增加服务器的可用资源。如果要处理的查询请求太多，则可能需要增加额外的副本来均衡查询负载。在以上任一种情形中，Solr 可能有次优的缓存性能或者查询语句中有错误，这些都需要反复调试。本节将会介绍如何理解 Solr 的内部性能统计数据 and 日志文件，获得调试所需的性能信息。

### 12.9.1 Solr 的插件 / 统计页

打开 Solr 的管理页面，选择一个 Solr 内核，在页面的左栏便会出现插件 / 统计 (Plugins/Stats) 链接，点击该链接可以查看 Solr 的插件 / 统计页。图 12.6 展示了从 Solr 的管理页面访问插件 / 统计页的效果。

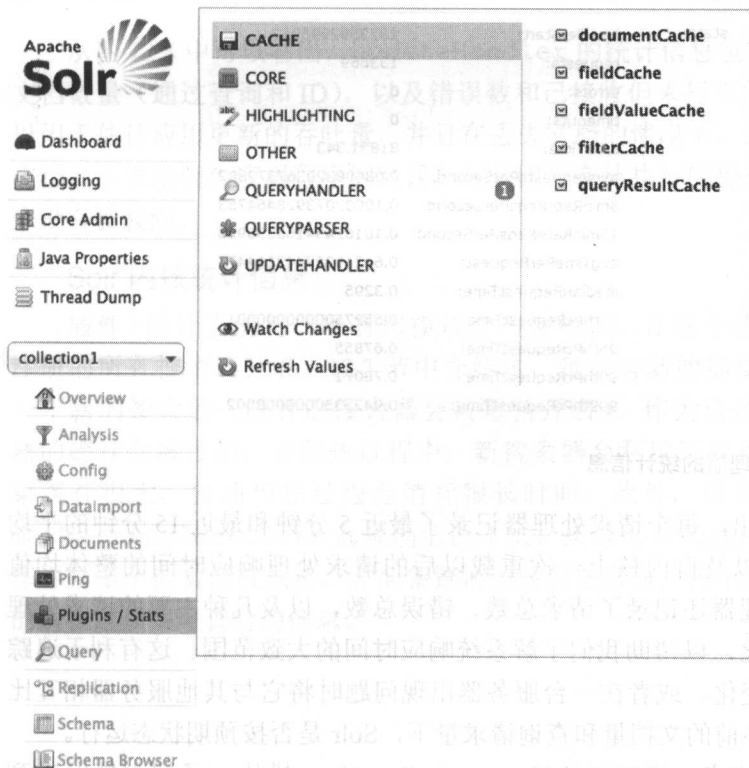


图 12.6 Solr 管理界面的插件 / 统计功能。这里可以查看有关缓存、内核、查询处理器及其他 Solr 组件与插件的情况

通过插件 / 统计页面可以了解 Solr 的大多数核心组件的性能特征。

### 查询和更新请求统计信息

如果要查看 Solr 的查询请求率及其历史响应时间，可以点击 QUERYHANDLER 链接，然后选择 /select 请求处理器（或者任何其他已定义的处理器），如图 12.7 所示。



图 12.7 查询相关请求处理器的统计信息

从图 12.7 可以看出，每个请求处理器记录了最近 5 分钟和最近 15 分钟的平均请求处理响应时间，以及自内核上一次重载以后的请求处理响应时间的整体均值和中值。同时请求处理器还记录了请求总数、错误总数，以及几种主要的请求处理响应时间所占的百分比，以帮助我们了解系统响应时间的大致范围。这有利于追踪 Solr 集群性能的历史变化，或者在一台服务器出现问题时将它与其他服务器相互比较，也有利于判断在当前的文档量和查询请求量下，Solr 是否按预期状态运行。

除了监控查询响应率，还可以监控 updateHandler 模块，了解文档发送到 Solr 中被索引的过程。图 12.8 展示了索引过程的相关信息。

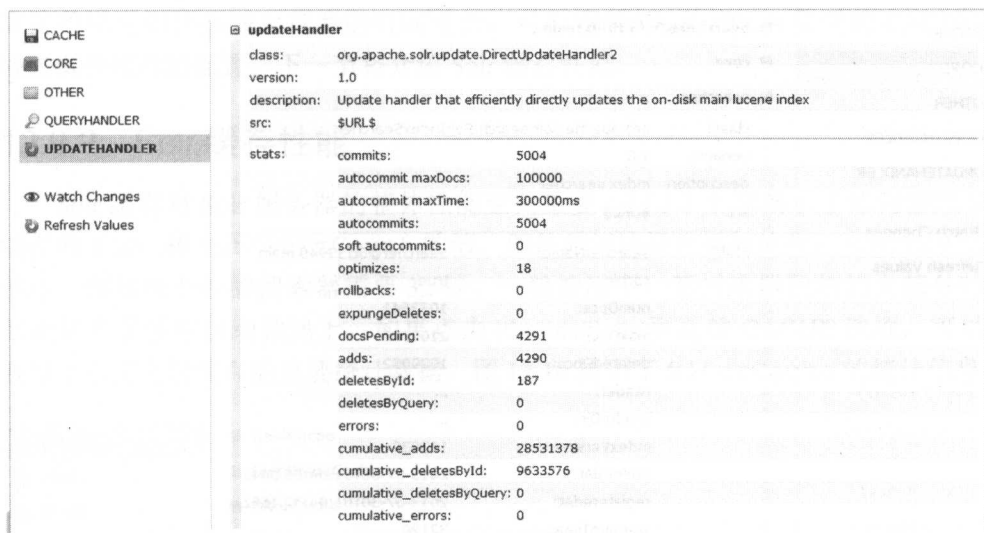


图 12.8 Solr 管理界面中更新处理器页面上 updateHandler 中有关索引的统计信息

从图 12.8 中可以看出，updateHandler 的统计信息包括添加、提交和删除的文档数量（通过查询和 ID），以及错误数和已接收但未提交的文档数。这些信息可以用于估计应用更新的吞吐量，并且在丢失文档的情况下，这些信息可以用于判断文档的丢失是因为应用程序的错误，还是因为有待执行的提交操作，又或是因为从来没有被收到。

### Solr 内核统计信息

插件 / 统计页的另一个有用模块是内核模块。在这个模块中可以查看当前已注册的所有搜索器列表。12.3 节中介绍过，每次有新的提交操作发生，就会开启一个新的搜索器（或者旧搜索器会被重新开启）。作为该过程的一部分，新搜索器的缓存会被预热，在预热过程中，新搜索器会利用旧搜索器中的一些数据。如果缓存很大，自动预热过程会消耗很长时间。此外，如果 solrconfig.xml 文件中 maxWarmingSearchers 选项的值比 1 大，可能会出现这样的情形：多个提交操作导致多个搜索器被预热，并准备置换当前的搜索器。图 12.9 展示了插件 / 统计页的内核选项中可查看的信息。



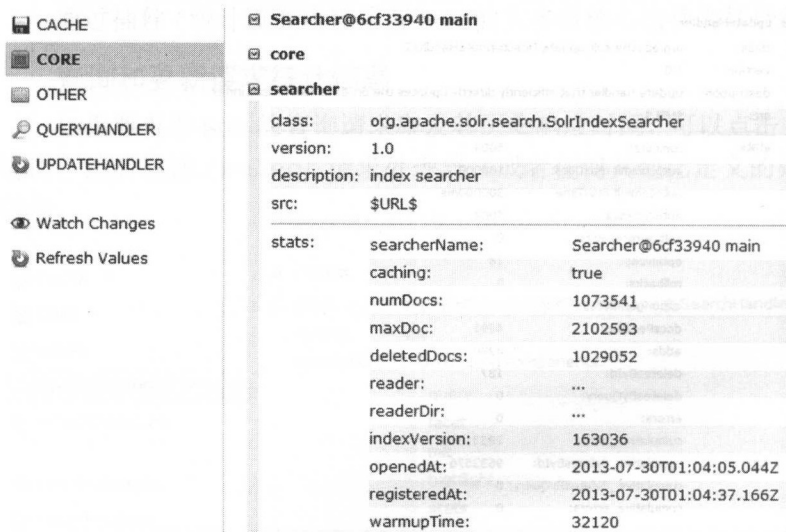


图 12.9 Solr 管理界面上 CORE 选项中有关搜索器的统计信息

当一个新的搜索器被加载并处于预热阶段时，可以在插件 / 统计页的内核选项中看到多个搜索器。变量 `searcher` 是对当前活跃的搜索器的引用（在图 12.9 中，底层实际的搜索器对象是 `Searcher@6cf33940`），无论正在预热的搜索器何时开始接收搜索请求，`searcher` 参数的引用始终都指向新的底层搜索器对象。任何发送给旧搜索器的请求都将被处理完，并且一旦对旧搜索器的所有查询和引用都被处理完（新的请求都被转向了新的搜索器），旧搜索器就会被丢弃。

管理页面中搜索器的重要统计数据之一是 `warmupTime`。在图 12.9 中，搜索器的 `warmupTime` 是 32 秒（32 120 毫秒），意味着自上一次提交操作发生之后，新搜索器花费了 32 秒来预热缓存并开始接收搜索请求。如果新搜索器的预热时间超过了两个提交操作之间的间隔，则 Solr 实例可能会陷入死循环，即搜索器缓存还未结束预热，新的搜索器又将进入预热状态，这会导致索引频繁地被置换，无法跟上新提交操作的步伐。在这种情形下，如果允许过多的搜索器同时预热，则 Solr 可能会内存不足，或者过度使用 CPU。因此，如果预热缓存的内容很多，那么保持 `maxWarmingSearchers` 参数为一个较小的值（比如 1）比较好。如果 Solr 实例关闭了可以快速索引文档的缓存（类似仅执行索引操作的服务器），则可以为 `maxWarmingSearchers` 参数设置一个较大值，以允许频繁的提交操作。

在图 12.9 的示例中，只要提交操作发生的间隔大于 32 秒（加上一些合理的缓冲），新搜索器预热应该就可以满足提交操作的频率。但是在每次提交操作后，新文档都需要过 32 秒才会出现在索引中，是否愿意接受这种延时只有读者自己才清楚。如



果觉得提交和预热缓存花费的时间过长，可以查看 Plugins/Stats 页面的 CACHE 模块来调整缓存的配置。下一节将介绍 Solr 缓存性能。

## 12.9.2 Solr 缓存性能

Solr 缓存对 Solr 服务器的整体性能至关重要。12.3.3 小节展示了如何修改 Solr 缓存的 size 和 autowarmCount 的值，但是如何选择合适的值呢？在图 12.9 的示例中，预热每个新的搜索器需要 32 秒。一般情况下，新搜索器预热的大部分时间都消耗在了缓存自动预热上。如果点击 Plugins/Stats 页面的 CACHE 选项，可以看到每个已定义的缓存的重要统计数据。图 12.10 展示了 filterCache 的统计信息。

**CACHE**

- CORE
- OTHER
- QUERYHANDLER
- UPDATEHANDLER
- Watch Changes
- Refresh Values

**fieldCache**

**fieldValueCache**

**filterCache**

class: org.apache.solr.search.FastLRUCache  
 version: 1.0  
 description: Concurrent LRU Cache(maxSize=100000, initialSize=100000, minSize=90000, acceptableSize=95000, cleanupThread=false, autowarmCount=1000, regenerator=org.apache.solr.search.SolrIndexSearcher\$2@6fbc969f)  
 src: \$URL\$

stats:	lookups:	4611
	hits:	4552
	hitratio:	0.98
	inserts:	58
	evictions:	0
	size:	1067
	warmupTime:	29944
	cumulative_lookups:	167278
	cumulative_hits:	165131
	cumulative_hitratio:	0.98
	cumulative_inserts:	2146
	cumulative_evictions:	0

图 12.10 filterCache 的统计信息，其中命中率高达 0.98，但是 warmupTime 达到 30 秒

图 12.10 中最重要的用于调整缓存配置三个统计信息是缓存的大小（1067）、命中率（0.98）和预热时间（30 秒）。图 12.10 和图 12.9 展示的是同一台 Solr 服务器的统计信息，搜索器总计的预热时间都是 32 秒，意味着过滤器缓存占据了总的搜索器预热时间的 30/32。

在调试缓存时，必须要平衡缓存命中率和达到该缓存命中率所需的预热时间（和最终使用的内存总量）。缓存命中率可以表明某条记录在缓存中被找到的次数占总的缓存查找次数的比例。缓存命中率的理想值为 1，表示所有的请求值都被缓存了；相反地，缓存命中率为 0 则表明缓存从来没有发挥过作用。缓存命中率为 0.5 以上

则意味着缓存在超过半数时间里对查询请求起到了帮助作用。如果缓存命中率较低,应当考虑完全禁用缓存(因为缓存并没有发挥多大作用,反而导致了额外的缓存检查操作),或者增加缓存的大小来提高缓存命中率(如果有效的话)。

在图 12.10 的例子中,缓存中存储了 1067 个过滤器,命中率为 0.98,这是一个接近完美的值。接下来需要考虑的问题是,减小缓存的大小是否仍然能够产生较高的缓存命中率。例如,缓存的大小被减小到 500,缓存时间可能会被缩短到 15 秒。这时缓存命中率呢?如果大部分的查询仅仅利用了前 500 个过滤器的话,缓存命中率可能不会有太大的改变。换句话说,在图 12.10 的例子中,98% 的查询总共利用了 1067 个过滤器,如果其中 96% 的查询总共只利用了前 500 个过滤器,甚至其中 90% 的查询总共只利用了前 50 个过滤器,那么就可以将预热时间从 32 秒减小为 1.5 秒,而缓存命中率仍然保持在 9/10 左右。

这个过程需要调整每一个缓存:计算出缓存的消耗(预热时间和消耗的内存)和缓存命中率的大小,然后决定如何通过增大或减小缓存的大小来最大化缓存命中率与缓存消耗的比值。

目前讨论的大部分关于监控 Solr 的示例都使用了 Solr 的管理页面的统计信息。接下来的小节将介绍其他获取 Solr 的监控信息的方法。

### 12.9.3 从请求处理器和 MBeans 获取统计信息

尽管前面的小节展示了如何从 Solr 的管理页面获取有用的 Solr 的性能信息,但请注意,那些页面实际上是利用 Solr 请求处理器获取信息的便捷方式。

例如,复制页的所有信息可以通过 Solr 的复制处理器获得,该处理器在 solrconfig.xml 文件中被定义,默认路径为 /replication。类似地,所有的缓存器、Solr 内核和请求处理器的信息都可以通过 MBeans 处理器获得。代码清单 12.5 展示了 MBeans 处理器的示例输出。

#### 代码清单 12.5 处理器的 Solr 性能统计

查询请求

```
/solr/core1/admin/mbeans?stats=true&wt=json
```

响应结果

```
{
  ...
  "solr-mbeans": [
    "CORE", {
      "core": {
```

```

...
"stats":{
  "coreName":"core1",
  ...
  "searcher":{
    ...
    "stats":{
      "searcherName":"Searcher@362c975a main",
      "caching":true,
      "numDocs":1069399,
      "maxDoc":1137777,
      "deletedDocs":68378,
      ...
      "indexVersion":163412,
      "openedAt":"2013-07-30T04:39:04.509Z",
      "registeredAt":"2013-07-30T04:39:26.564Z",
      "warmupTime":22053}},
    ... }}}},
"QUERYHANDLER",{
  ...
  "standard":{
    ...
    "stats":{
      "requests":238083,
      "errors":0,
      "timeouts":0,
      "totalTime":2.0561250266E7,
      "avgRequestsPerSecond":17.51375685220339,
      "5minRateReqsPerSecond":20.355573970348654,
      "15minRateReqsPerSecond":18.66862341791267,
      "avgTimePerRequest":86.36241558965226,
      "medianRequestTime":75.787,
      "75thPcRequestTime":99.09899999999999,
      "95thPcRequestTime":166.72789999999995,
      "99thPcRequestTime":260.87716000000006,
      "999thPcRequestTime":869.9387500000016}}
    ...}}}

```

按照每个 Solr 内核组织信息。

搜索器统计，例如，预热时间。

搜索器统计，例如，预热时间。

查询响应时间，错误及相关统计信息。

代码清单 12.5 的数据应该看起来十分眼熟，它与 12.9.1 节和 12.9.2 节中的 Solr 管理页面中的数据属于同一类。这样可以更加容易从任何地方，甚至是 Solr 应用程序的外部，程序化地导入服务器性能信息。

### 12.9.4 外部监控选项

除了可以从 Solr 的 MBean 处理器获得 MBean 的信息，还可以使用外部监控工具利用 JMX 获取信息。回忆一下第 4 章的内容，JMX 是一个协议，使得系统信息可以通过网络被获取，以用于管理和监控。要启用 JMX，首先需要在 `solrconfig.xml` 文件中添加一行代码：

<JMX/>

一旦 JMX 在 Solr 中被启用，则在启动 Solr 时就需要添加下列额外的参数，以告知 JVM 允许 JMX 参数被远程获取。

```
java -Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9001
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
-jar start.jar
```

这些设置都是可以根据 JMX 监控方案的配置来进行配置的。如果没有使用默认的 Jetty 配置 (java -jar start.jar)，还需要单独配置 Java servlet 容器或者启动设置，以确保这些额外的 JVM 参数会被启用。大部分现代应用程序性能监控工具都能够读取 JMX beans 并提供长期的度量集合和图表，通常伴随着监控，并在数值偏离预设的性能标准太远时给出警告。除此之外，一些应用程序性能监控工具——包括基于云的工具——能够直接支持和理解 Solr 的内部。如果你简单地在互联网上搜索一下 Solr application performance monitoring，将会找到一长串提供 Solr 集群性能监控的公司的名单。

### 12.9.5 Solr 日志

对于大多数的应用程序来说，日志文件是了解任意时刻 Solr 集群的状态的最丰富的信息源。它们记录了运行过的查询请求、每个请求的全部参数、每个请求执行的时长，以及应用程序的所有错误或发生在请求期间或不同请求之间的重要操作。Solr 的日志文件还提供了文档何时被收到、提交操作和块合并何时发生、复制何时开始和结束，以及所有被定义在日志配置中的细节层次的其他操作的详细信息。

### 12.9.6 加载测试

使用实际流量测试是确定 Solr 集群能力的最佳方式之一，此外进行离线加载测试也是确定 Solr 集群能力的好方式。例如，重放历史日志就是一个模拟用户流量，进而判断服务器的查询和索引能力的很好的方法。

开源项目 SolrMeter 提供了强大的专为加载测试设计的特性。SolrMeter 允许指定将哪些查询或者文档发送到 Solr 中，以及发送的速度。接下来，SolrMeter 会评测用户请求各个部分的性能、缓存的利用情况和当时 Solr 集群中发生的具体细节（例如提交或者优化）。图 12.11 展示了一个查询负载与查询响应时间关系的基本柱状图，这是 SolrMeter 提供的 Solr 性能的众多衡量标准之一。

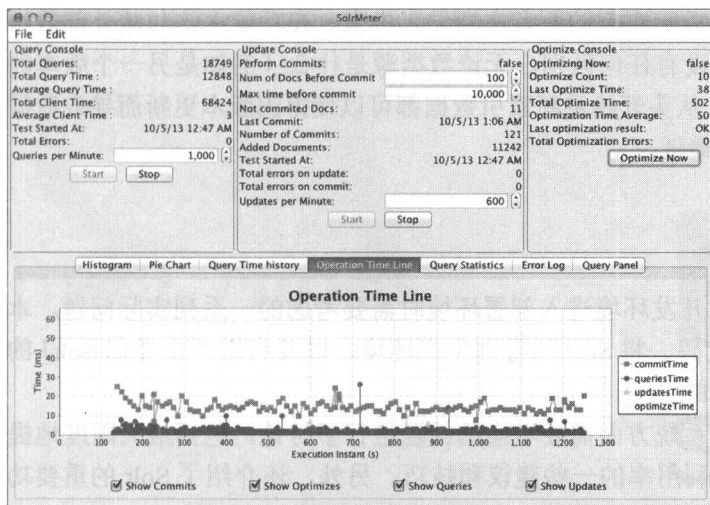


图 12.11 Solr 的负载测试工具 SolrMeter 截图

除了图 12.11 中的柱状图，SolrMeter 还提供不同查询处理时间范围的饼状图，以及运行中出现的错误列表，甚至还提供命中率、查询次数和测试时每个缓存的置换数。如果想要了解如何快速测试 Solr 应用程序的能力，可以访问 SolrMeter 的网站 <https://code.google.com/p/solrmeter/>。

## 12.10 不同Solr版本之间的升级

Solr 一直都处于持续的开发状态，这意味着你时不时地需要将 Solr 从一个较老的版本升级到一个新版本，以获得最新最强大的功能并提升性能。升级 Solr 最好的办法就是安装新版本的 Solr，对配置文件和插件做必要的定制，并在新版本下通过从外部源添加所有内容重新构建索引。

从技术上讲，Solr 的每个官方发布版都需要支持自动更新之前的主要版本的索引文件。这意味着如果有一个基于 Solr 3.x 构建的索引，在安装了 Solr 4.x 并且利用旧索引启动之后，Solr 应该能够自动更新之前的索引文件。如果采取这种方式，你可能就希望在启动新版的 Solr 之后，执行索引优化以确保成功地更新了所有的索引文件。

但是，这并不是一个好的更新索引文件的方式。各种文本分析组件会随着 Solr 的版本变化而变化（即使是很小的版本变化），这意味着在新版本下构建的索引中的分词可能与之前版本的不同。不幸的是，索引格式的更新无法克服这种局限，这意味着在索引和查询时间分析之间会有一些不匹配，这会导致数据完整性的问题。

如果恰好在索引中存储了所有的字段，可以在新版本的 Solr 中使用 DIH 之类

的工具来把旧的运行中的 Solr 实例的所有文档导入进来。在功能上这与从外部数据存储中重新导入所有文档没有任何区别。无论数据源是什么（例如是另一个版本的 Solr 或某些其他数据源），从头开始重新索引数据都可以使得因版本更新而导致数据完整性的问题的出现概率最低。

## 12.11 本章小结

本章旨在介绍 Solr 从开发环境进入部署环境时需要考虑的一系列实际问题。本章内容包括编译 Solr 源代码、将 Solr 部署到生产环境，以及利用一个类似 SolrJ 的客户端库向 Solr 发送请求的过程。

本章也对硬件和操作系统方面需要考虑的问题进行了讨论，包括最大限度地提高内存和其他系统资源的利用率的一些建议和技巧。另外，还介绍了 Solr 的重要功能特征，以及如何监控、调试和测试 Solr 中的许多相关配置。

本章介绍的一个重要主题是如何通过分片和将数据复制到多台服务器来扩展 Solr。扩展 Solr 实际上是服务器管理和配置管理的一大难题，因此下一章将介绍如何在 Solr 中大大降低这方面的复杂度。

## 第3部分

# Solr进阶

随着对 Solr 的深入了解，你应该也想尝试用 Solr 解决更复杂的搜索问题了。在接下来的 4 章中，我们将介绍一些具有挑战性的高级主题，这些内容来源于许多世界级搜索应用的经验总结。你可能不会对每个主题都感兴趣，但今后有可能会经常遇到它们。这几章提供了相应的处理方法。

首先登场的是 SolrCloud，它是一组分布式功能，在集群中可以对索引轻松地进行分片和复制。第 13 章的内容相对理论化一些，但读完它有助于你了解如何使用 SolrCloud 来实现可扩展性和高可用性需求。

世界正变得越来越小，越来越多的组织在进行全球化发展，多语种的处理成为许多搜索应用的常见需求。在 Solr 中如何处理多语种内容呢？第 14 章会全面介绍多语种搜索的基础知识和高级技术。

Solr 在查询阶段能对复杂数据进行一系列的稳健操作，这是第 15 章的主题。具体来说，我们将介绍动态字段值生成的各种方法，包括函数查询（适用于结果返回、排序和排名）、多维度的分面和高级数据分析、地理空间搜索、跨文档和索引的连接，以及引用外部数据到 Solr 的索引。

对组织而言，很常见的做法是从基础的关键词搜索开始，然后根据专业领域知识对文档排名进行微调。如何实施专家级别的相关度模型是第 16 章的重点。第 16 章的大部分内容源自作者在 Solr 中开发个性化推荐系统的实践经验总结，这些内容也说明了在技术应用层中不应将 Solr 仅定位于一个关键词搜索引擎。



# 13

## SolrCloud

---

### 本章要点

- 借助 SolrCloud 架构扩展 Solr
- 用 ZooKeeper 管理配置信息
- 分配索引和查询
- 管理 SolrCloud 系统
- 分片和自定义散列

在本章，你将学习如何使一组功能去设计、配置和操作大规模的 Solr 集群，这里把这组功能统称为 SolrCloud。由于本章包含一些你可能不太熟悉的新概念和专业术语，所以学习起来会有一定挑战性。不过请放心，读完本章，你就能掌握如何管理 Solr 集群，以及对如何搭建并运行强大的大规模分布式搜索引擎有所了解。

本章内容偏理论，上手操作相对少一些。你会发现启用 SolrCloud 模式是一件很容易的事情。索引和查询的现有客户端代码无须修改。从客户端应用程序角度看，SolrCloud 集群和其他 Solr 服务器是一样的。

为了解释清楚 SolrCloud 的核心概念，本章使用一个名为 logmill 搜索引擎示例，用来支持日志聚合和分析服务。这个虚构的应用程序从许多系统中收集日志信息，将其聚合到一个集中式的搜索引擎，对应用程序的活动进行监控、数据可视化和分析。

SolrCloud 非常适合解决此类问题。因为它支持近实时索引，可以处理上百个应用所产生的大量日志信息。实际上，许多查询都比较复杂，例如，分析日志指标的分面。

本章介绍 SolrCloud 的主要概念。当需要解释比较复杂的概念时，我们就会以 logmill 这个简单示例来讲解。之所以选择它，是因为它在概念上足够简单，并且在应对当今大型组织的 Web 应用和企业产生的大量信息时，它具备可扩展性。

## 13.1 SolrCloud上手

在了解 SolrCloud 架构之前，首先在本地计算机上启动 SolrCloud 集群。这样做可以先接触到一些核心概念，在实际操作中逐步理解更加复杂的概念。

### 13.1.1 在云模式下启动 Solr

本节为一个 Solr 集群启用日志聚合服务，即 logmill。具体来说，第一个 Solr 集群包含跨本地工作站上两个 Solr 实例的索引分割。每个索引分割称为分片，每个分片包含 logmill 应用的大约一半的文档。

在 SolrCloud 中，跨多个节点的索引分割称为集合（collection）。迄今为止，本书只提到过 Solr 的内核（即一个点只有一个索引），还没遇到过 Solr 集合。在 13.2.1 节中会深入讲解集合这一概念。目前，你只需要知道，集合是 Solr 中跨多节点的索引分割。在本章的示例中，我们把集合称为 logmill。

SolrCloud 还依赖于分布式协调服务，即 Apache 的 ZooKeeper，在 13.2.2 节中会介绍相关内容。目前，你只需要将其看作一项抽象服务，该服务能管理集群状态，将配置文件分配到集群中的各个节点上。为简单起见，本章将 ZooKeeper 内嵌在 Solr 的同一个 Java 进程中。

#### 创建 logmill 内核

现在我们打算创建第一个 Solr 集群，首先将 example 目录下的内容复制到名为 shard1/ 的新目录下。请注意，\$SOLR\_INSTALL 是顶层目录，第 2 章介绍过，要将 Solr 安装文档解压缩到这里。此步骤是必须要做的，因为我们需要通过绑定不同的端口来实现在一台计算机上运行多个 Solr 实例。

```
cd $SOLR_INSTALL/  
cp -r example/ shard1/
```

接下来，复制 shard1/Solr 目录下面的 collection1/ 目录，为 logmill 创建一个内核目录。要确保 data 目录为空，不包含任何数据和索引文件。然后，删除内核目

录中已有的 `core.properties` 文件，确保其中的内核不会自动被添加到 SolrCloud 集群。最后，将 `core.properties` 文件中的内核名字设为 `logmill`，从而启用内核的自动发现功能。

```
cd shard1
cp -r solr/collection1/ solr/logmill/
rm -rf solr/logmill/data/
find . -name "core.properties" -type f -exec rm {} \;
echo "name=logmill" > solr/logmill/core.properties
```

Solr 4.4 的一个新功能，启用 Solr 内核的自动发现功能。

现在就可以在云模式下启动 Solr 了。

### 在云模式下启动 Solr

第一次启动 SolrCloud 集群时，需要先启动一个 Solr 节点，此节点是其他所有节点启动的基础。本书把这样的第一个节点称为引导节点（bootstrap node），因为它为集群的其余部分执行了特殊的一次性初始化工作。当然，这些特殊的一次性操作都相当简单。

执行代码清单 13.1 中的命令，在云模式下启动 Solr。

代码清单 13.1 在云模式下启动 Solr

在 SolrCloud 中创建 logmill 集合。

```
java -Dcollection.configName=logmill
-DzkRun
-DnumShards=2
-Dbootstrap_confdir=./solr/logmill/conf
-jar start.jar
```

运行 ZooKeeper，与 Solr 一样，内嵌相同的 JVM。

上传 logmill 配置文件到 ZooKeeper。

将 logmill 索引分成 2 个分片。

以上代码将在云模式下开启 `shard1` 的 Solr 服务器。让我们来仔细分析一下启动 SolrCloud 集群指定的 `-D` 参数。`collection.configName` 参数指定了 ZooKeeper 配置目录的名称，在此例中为 `logmill`。在 SolrCloud 中，每个集合都需要在 ZooKeeper 中确定一个已命名的配置目录。例如，`logmill` 集合使用名为 `logmill` 的配置目录。同一个集群可以托管多个集合，但目前我们只会使用到一个集合。`zkRun` 参数告诉 Solr，在同一个 JVM 中启动一个嵌入式 ZooKeeper 实例。该嵌入式 ZooKeeper 实例的监听端口是 9983。`numShards` 参数应是不言自明的。前面提到，在本例中，我们决定把 `logmill` 集合分配到两个分片上。

`bootstrap_confdir` 参数告知引导节点将其配置文件加载到 ZooKeeper。在 SolrCloud 中，ZooKeeper 的主要功能之一是集中化的配置储存。集中配置允许集群中的所有节点从中心位置直接下载配置文件，而不需要等系统管理员将配置更改推送给多个节点之后再下载。在 ZooKeeper 能提供集中化配置文件之前，需要提

前将配置文件加载到引导服务器上。

使用 Web 浏览器访问 <http://localhost:8983/solr>，打开 Solr 的管理控制台。图 13.1 显示了在云模式下运行 Solr 时，云（Cloud）面板被激活的状态。

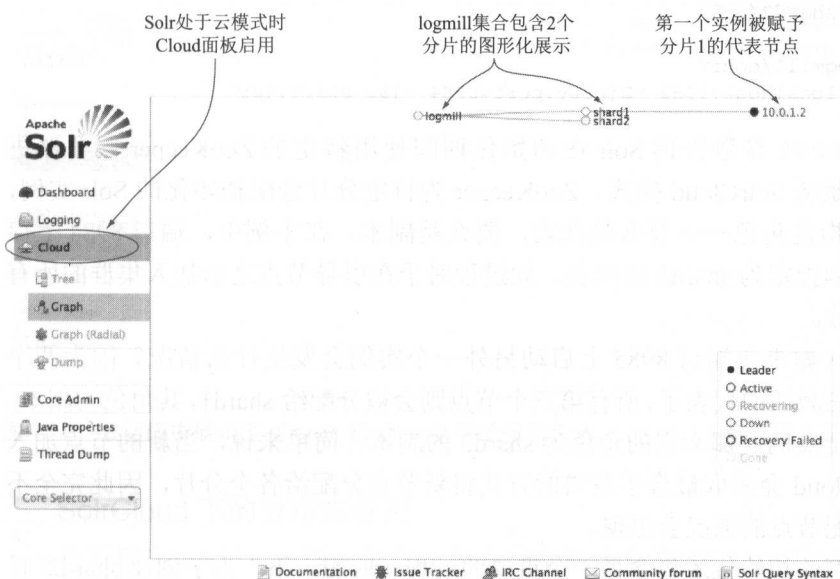


图 13.1 云面板显示了包含两个分片的 logmill 集合的图形化视图。本章启用的第一个 Solr 实例是 shard1 的代表

请注意，图 13.1 有两个分片：shard1 和 shard2，但是只有 shard1 被分配了一个节点。这是为什么呢？因为此时只启动了一个 Solr 服务器。接下来将会为 shard2 启动另外一个 Solr 服务器。

还需要注意的是，如图 13.1 中的黑点所示，第一个节点是 shard1 的代表。下一小节会详细介绍分片代表（shard leader）。这里简单提一下，当分片代表处理更新请求时，可以处理一些其他任务。例如，为新建或者更新的每个文档分配唯一的版本号。

与书中多次提到的查询做法一样，选择 logmill 内核，执行匹配所有文档的查询（\*:\*）。这时 Solr 会返回 503 错误，显示错误信息为“无服务器托管分片”（no servers hosting shard）。查询失败的原因是，在配置 Solr 时将 logmill 的索引分割成了两个分片（-DnumShards=2），但目前只有一个运行实例的分片。请记住，分布式查询的使用前提是，每个分片都要包含激活的服务器，否则不能向 Solr 提交查询请求。

### 为 shard2 启动另一台 Solr 服务器

接下来，我们将启动另一台 Solr 服务器来托管 shard2。你应该还记得，Solr 的

默认端口是 8983。要在计算机上启动另外一台服务器，需要绑定另外一个端口，如 8984。回到命令行窗口，执行以下命令：

```
cd $SOLR_INSTALL/  
cp -r shard1/ shard2/  
cd shard2/  
rm -rf solr/logmill/conf/  
java -DzkHost=localhost:9983 -Djetty.port=8984 -jar start.jar
```

其中，zkHost 参数告诉 Solr 在初始化期间使用特定的 ZooKeeper 服务器进行注册，从而激活 SolrCloud 模式。ZooKeeper 为特定分片分配初始化的 Solr 实例，并且会为分片指定角色——要么是代表，要么是副本。在本例中，端口 8984 上启用的第二个实例指定为 shard2 的代表。此过程对于在引导节点之后加入集群的所有其他节点都有效。

思考一下，如果在端口 8985 上启动另外一个实例会发生什么情况？因为两个分片都已经存在激活的代表了，所有第三个节点则会被分配给 shard1，其角色为副本。如果启用第四个实例，那么它的角色为 shard2 的副本。简单来说，当新的节点加入集群时，SolrCloud 会采取最合乎逻辑的方式将新节点分配给各个分片，因此完全不用担心手动分配节点的情况会出现。

第二个实例的初始化还包含另一个重要的过程。回想一下，为了创建 shard2/ 目录，我们复制过 shard1/ 目录。当时还专门删除了 shard2/solr/logmill/conf/ 目录下的内容，以确保在引导节点之后，新加入集群的节点都能从 ZooKeeper 处接收到配置文件，而不是来自本地文件系统。

现在两个实例都运行起来了，现在再看看管理控制台的云面板。图 13.2 显示有两个分片的 SolrCloud 集群，一台主机托管一个分片。启动第二个服务器之后，一定要刷新页面。

请注意，Solr 自动将运行在端口 8984 上的第二台服务器分配给了 shard2。这里还有另外一种选择，即把新节点分配给 shard1，作为 shard1 的副本。然而，在新节点加入集群且被分配副本角色之前，Solr 要确保所有的分片至少有一台托管主机。两台服务器都应该是绿色，即为激活状态。因为每个分片都只有一台主机，所以两台服务器都是代表，图中用黑点表示。图 13.2 中的集群状态信息会储存在 ZooKeeper 中。

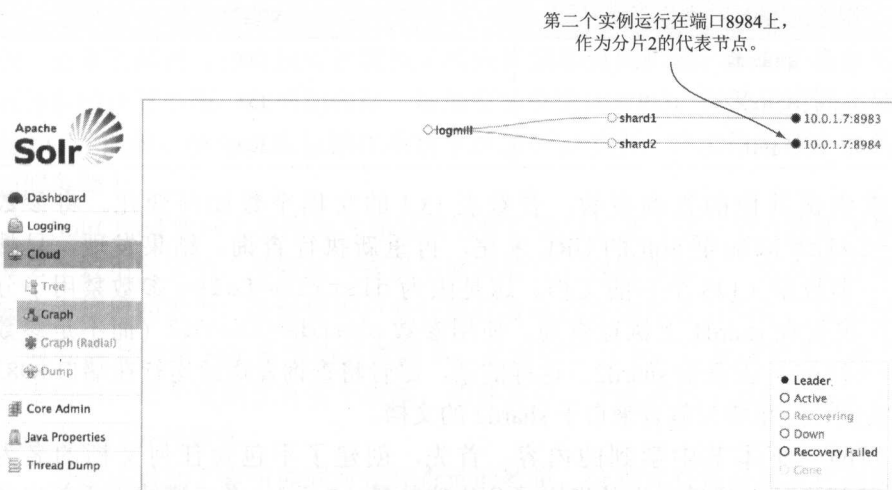


图 13.2 云面板中展示了 shard1 和 shard2 中的代表角色：logmill 集合已在云模式下全面运行

### SolrCloud 下的分布式查询

重新在 logmill 内核下执行匹配所有文档的查询 (\*:\*), 现在就可以成功地执行该查询了。由于还没有在 logmill 集合里添加任何文档, 因此结果显示为零。这里通过添加几个示例文档来解决此问题。

由于虚构的 logmill 搜索应用是针对日志文件的, 我们使用随书附带代码提供的的一个简单工具, 对 Solr 的日志信息示例文件进行索引。

```
cd $SOLR_IN_ACTION
java -jar solr-in-action.jar indexlog -log=example-docs/ch13/solr.log
```

indexlog 工具能够解析指定的 solr.log 文件, 并且能够为每条日志信息索引一个新文档。运行 indexlog 之后, 重新执行匹配所有文档的查询 (\*:\*), 就会看到 logmill 索引包含了 300 个文档。SolrCloud 对文档进行索引时, 会根据其默认的唯一 ID 的哈希值将其发送到其中的一个分片上。13.3 节会详细介绍文档如何获得发送路径。一个文档仅能存在于唯一的分片中。每个分片占用独有的散列区间, 散列函数会试图在分片之间将各个文档进行均匀分配, 所以集群中每个分片的大小大致相同。13.3 节会加详细介绍分布式索引。表 13.1 是运行 indexlog 之后的文档分配情况。

表 13.1 由 indexlog 生成的文档分配情况

分片	文档数
Shard1	143
Shard2	157
总计	300

接下来尝试其他的查询参数，看看表 13.1 的文档个数如何变化。将参数 `distrib=false` 添加至 Solr 的 URL 末尾，再重新执行查询。结果发现，只搜索到大约一半数量（143 个）的文档。这是因为 `distrib=false` 参数禁用了分布式搜索，仅仅在 `shard1` 上执行查询。使用参数 `shards=shard2`（而不是参数 `distrib=false`）去查询 `shard2`。这样的话，尽管将查询发送给运行在端口 8983 上的 Solr 实例，结果集只包含来自于 `shard2` 的文档。

简要回顾一下本节中学到的内容。首先，创建了不包含任何文档的名为 `logmill` 的新内核。然后，为其指定了分片的数量（2 个），在云模式下重新启动 Solr，启用了 ZooKeeper 的内嵌实例。接着，用 `numShards` 参数开启了第一个实例，即引导节点。为了能够在 SolrCloud 下执行查询，每个分片都必须自己的处于激活状态的服务器。当执行分布式查询时（默认为云模式下），查询会发送到所有分片中，最终结果集会包含来自于所有分片中的文档。另外，根据文档唯一 ID 值的散列函数，可以将文档只分配到其中一个分片上。现在，SolrCloud 集群运行起来了，下一节介绍 SolrCloud 架构背后的动机。

13.1.2 SolrCloud 架构的驱动因素

13.2 节会进一步讲解一些核心概念，比如集合、ZooKeeper、分片代表和集群状态管理等。在进入实施的具体细节之前，需要了解 SolrCloud 架构的特点。SolrCloud 架构驱动的分布式搜索引擎具有 5 个主要特性：

- 可扩展性
- 高可用性
- 一致性
- 简单性
- 灵活性

接下介绍这 5 个特性，详细了解如何将这特性应用于通用的以及 Solr 这样的大型搜索引擎。



## 可扩展性

可扩展的软件系统能够通过在一个或者多个维度上扩容来处理增加的工作负载。通常有两种方法实现可扩展性：横向扩展和纵向扩展。纵向扩展包括增加单个服务器的计算资源，如增加内存、增加更多更快的 CPU，以及使用固态硬盘提升磁盘 I/O 性能等。横向扩展包括在系统里添加更多节点，将工作负载分配到多台并行的服务器上。

SolrCloud 倾向于使用分片和复制这样的横向扩展方式。分片能将大型索引分割成多个较小的索引。尽管这些索引不能安装在一台服务器上，但也能允许用户同时操作大规模的索引，还能使用户并行执行复杂的查询和索引操作。复制操作能在多台服务器上创建 Solr 索引的其他副本，从而增加冗余，防止故障发生时造成损失。在索引中，每个文档都有多个副本，即使其中一个副本出了问题，Solr 也能借助其他副本执行各种操作。

复制还增加了索引能同时执行查询的数量。分布式查询会被发送到每个分片的一个副本上。假设每个分片有 10 个副本，就能同时执行 10 倍之多的查询。13.4 节会介绍分布式查询的有关内容。

扩展 Solr 这样的系统都是为了让它可以线性扩展，即使用更多资源线性地增加系统的计算能力。例如，如果集群中的节点数量增加一倍，线性扩展意味着系统能承担两倍的工作负载。但是，增加越多的节点之后，管理这些节点的额外开销也会随之增加，因此在实际情况中只能以实现接近线性扩展为目标。13.3 和 13.4 节中将会讨论分布式索引和查询的系统开销问题。

在纵向和横向扩展这两种方法中，横向扩展更受欢迎。因为它更具有成本效益，这主要得益于低成本的商用硬件和按需出现的云计算服务（如亚马逊的 EC2）。尽管 SolrCloud 主要侧重于横向扩展，但是如果认为仅仅依靠往 SolrCloud 集群里增加更多的机器就能实现所有的扩展性需求，那么就大错特错了。特别地，Solr 最适合于快速、多核的 CPU，容量大的内存和高速磁盘。实际上，大多数大规模的 Solr 安装都会用到上述两种扩展方法。例如，为了实现最优查询性能，需要足够的内存将索引缓存在操作系统缓存中，还需要内存来容纳 JVM 中的缓存、排序和分面。如果索引中可搜索的数据结构需要 20GB 的内存，那么还需要分配 8GB 内存给 JVM，算下来 Solr 大概需要 28GB 的内存。Solr 也可以是磁盘 I/O 密集型（尤其是为文档构造索引时），因此本书也推荐使用像固态硬盘这样的高速磁盘。

扩展搜索引擎并没有一个放之四海而皆准的问题解决办法，不同的维度需要用不同的策略去正确对待。表 13.2 总结了搜索引擎中可扩展性方面的常见问题，以及 SolrCloud 的应对策略。

表 13.2 搜索引擎中的可扩展性问题，以及 SolrCloud 的应对策略

可扩展性 / 限制	应对策略
索引文档数量：包含大量文档的索引会影响分面、排序和构造过滤器的性能。另外，由于文档的 ID 号是以整数型存储的，目前 Lucene 索引的文档数量限制为约 21 亿	使用分片策略将大型索引分割成多个较小的索引；分片的有关内容参见 12.5 节
文档大小和复杂性：具有很多字段或者大型文本字段的文档需要更多内存和更快的磁盘输入 / 输出	增加更多内存和高速磁盘
索引吞吐量：也许需要每秒为成千上万个文档构造索引	使用分片策略在跨节点平台上进行分布式索引操作
文档波动性：如果现存文档更新频率过快，索引也会具有很高的易变性，需要不断进行片段合并	使用高速磁盘有利于经常进行的片段合并；关于片段合并的内容请参见 12.3.3 节
查询量（通常以 QPS 来衡量，即每秒查询量）	借助复制操作增加执行查询时的可用线程数
查询的复杂程度：包括分面、分组、自定义排序影响和执行查询的性能	利用分片和复制操作并行化处理复杂的查询计算，例如，分面和排序

高可用性

满足了可扩展性要求之后，就需要为各种可预料和不可预料的故障做好准备，其终极目标是实现零宕机。现实中要实现此目标的成本异常高昂，而且这也超出了大多数组织的能力范围。一个合理的折中办法是面向高可用的商业决策进行系统搭建，而不是局限于技术本身。搜索引擎是否具备高可用性，这取决于经费投入，不要被架构的一些固有问题所限制。

系统的高可用性包含两个主要特性：关键业务失效时备援到正常服务上的能力和数据冗余性。具有数据冗余的系统在发生故障时，就不需要将像搜索索引这样的大数据集迁移到正常运行的节点上。即使在高速网络上迁移 100GB 的数据也很花费时间。一般来说，在使用像 Solr 这样的分布式系统时，需要为以下 4 种故障做好准备工作：

- 1. 由于硬件故障和断网这样的问题造成的意外中断，会影响集群里节点的子集。
- 2. 由于系统升级和系统维护任务而发生的计划内中断停机。
- 3. 由于系统负载较重，需要进行降级服务。
- 4. 造成整个集群或者数据中心脱机的灾害性事故。

SolrCloud 能够对同一个数据中心里的不同服务器进行故障转移和数据冗余，这一点就能解决前三种故障。但是，要将 Solr 分配到多个位置不同的数据中心上还有其他工作要做，这一点如果成功的话，就能解决第四种故障，即灾难事故恢复。自 Solr 4.7 起，没有为处于多个数据中心的集群 Solr 提供的内置支持。跨不同的数据中心分发节点，可以这样做，所有节点之间的通信都通过 HTTP 进行。但就实际情况来说，由于数据中心之间的网络延迟太高，很难实现可接受的搜索性能。也就是说，

对于运行在不同数据中心里的两个独立 SolrCloud 集群来说, Solr 没有内置任何功能来帮助复制数据。只能手动进行设置。

就基本面来说, Solr 提供两种服务: 创建索引和执行查询。这两种服务中的一种或者两种都可能需要具备高可用性。有人可能会认为, Solr 必须随时对查询作出响应, 但可以接受创建索引时的短暂停机。也有人可能认为, 两种核心服务都不能存在任何中断或者故障。SolrCloud 的一个关键的设计原则是, 集群里的每个节点都能创建索引和执行查询。与主从架构相比, 在主从结构中主节点担负构建索引的任务, 从节点负责执行查询; 而 SolrCloud 能同时实现两种核心服务的高可用性。

为确保 Solr 能继续提供查询服务, SolrCloud 通过让每个分片拥有多个副本的方式来支持数据冗余。为确保 Solr 能不断支持更新请求(创建索引), 在当前 leader 发生故障时, SolrCloud 可以选择另外一个新分片上的代表来自动转移故障。

Solr 还能在升级过程中通过滚动重启的方式最大限度地减少停机时间, 其中, 每台 Solr 服务器仅使用稳定的、基于 HTTP 的 API 与其他 Solr 的服务器通信。因此, 可以在同一个集群里暂时运行不同版本的 Solr, 例如, 执行滚动升级。

另一种故障类型是, Solr 处于联机状态并且能对查询做出响应, 但由于工作负荷很重, 无法对用户请求做出快速响应, 因而达不到其服务水平协议(Service-Level Agreements, SLA)的要求。这里想要说明, 即使服务器还能运行, 但不能快速响应查询请求的搜索引擎的使用效果会大打折扣。造成这种问题的 Solr 常见故障之一是, JVM 的完整垃圾回收器出现暂停现象。这种故障会妨碍其他线程的执行。通过可靠的监测系统可以预防这种故障的发生, 并且要在发生时快速地向集群添加更多的节点, 以应对增加的工作负载。SolrCloud 允许用户在集群中快速添加副本以满足不断增长的需求。

另外, 通过构造更具有容错性能的硬件系统来预防故障发生。这方面的讨论不属于本书范畴, 读者可以与系统管理员讨论一下, 如何能在硬件方面提升 Solr 中节点的容错性, 这样做是很有价值的, 例如, 对磁盘阵列进行 RAID 配置。

## 一致性

CAP 定理为理解分布式系统中的权衡问题提供了理论模型<sup>1</sup>。该定理指出, 在分布式系统中, 人们期望达到的性能有三个——一致性(C)、可用性(A)和分区容错性(P), 但是只能同时实现其中两个性能。实际上, 分布式系统设计人员假定故障一定会发生, 分区容错性则是必须要实现的性能。这样, 设计人员就要在一致性和可用性之间作出选择<sup>2</sup>。

1 CAP 定理也叫做布魯爾定理, 更多内容请参见 [http://en.wikipedia.org/wiki/CAP\\_theorem](http://en.wikipedia.org/wiki/CAP_theorem)。

2 “CAP 困惑: 分区容错性的问题”, Henry Robinson, 2010 年 4 月 26 日, <http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/>。

一致性是指操作要么成功，要么失败。像 Solr 这样的分布式系统中，一致性问题可以归结为，在写入数据后可以读出什么数据来。SolrCloud 假设一次写入成功之后，所有处于激活状态的副本都将返回文档的相同版本。这意味着，在所有副本上的更新请求必须全部成功，否则请求失败。例如，如果 shard1 有 A、B、C 三个副本，必须要三个副本都更新成功之后，此次的写操作才算成功。如果只有对 A 和 B 的更新成功执行，而对 C 的更新操作失败了，那么此次的写操作失败。

13.3 节会介绍更多有关分布索引的知识。一致性的关键在于，SolrCloud 不允许参与查询的副本存在相同文件的版本差异。可能因为写操作失败，该索引并没有完全更新，但是根据识别哪一个副本参与查询，从而控制该副本使其不会返回不一致的结果。

比起写操作的可用性来说，Solr 更强调一致性，并且不像一些 NoSQL 技术那样不能协调。在 Cassandra（一种 NoSQL 数据库）上执行写操作，用户可以指定所需的一致性程度，从而更加强调一致性或者写操作的可用性。

Solr 4 没有一致性程度配置，写操作必须在分片的所有处于激活状态的副本上执行成功。但是，Solr 确定一次写操作是否成功时，只会考虑处于激活状态和处于恢复过程中的副本，不会考虑离线副本也需要更新的问题。因为一旦使用内置的恢复过程对故障副本进行恢复，离线副本的问题就会被解决。简而言之，只要每个分片上都有一台活跃主机，Solr 就会一直接受写操作。

比起一致性来说，更偏向于写操作可用性的系统会接受对分片上任何副本进行的写操作，接受写操作的副本会将更新发送给同一分片中的其他副本。如果更新在其中一些副本上失败了，系统仍然会认为此写操作是成功的。为了达到写操作的高可用性，一些系统会允许客户端应用程序指定它们容忍这种弱一致性。这就是众所周知的最终一致性，即副本最终状态是一致的。目前 SolrCloud 对于这种弱一致性是零容错，即写操作在分片的所有副本上都必须执行成功。

比起写操作的可用性，为什么 Solr 更强调一致性？这是因为在编写客户端应用程序时，一致性能简化 API。用于创建 Solr 索引的客户端代码不需要担心部分失败，因为在分片的所有副本上，更新请求要么全部成功，要么全部失败。另一方面，只要查询同一个分片的内容，无论哪个副本参与其中，每次查询都能确保得到一致的搜索结果。试想一下，本书第 1 章的房地产搜索示例允许副本存在不一致的情形，那么可能出现一位用户能看见新房源的具体信息，而另一位用户却看不到的情况。由于参与查询的副本不同而导致应用程序提供了不一致的搜索结果，要树立和维护搜索应用程序的客户信任度的话，可想而知是异常困难了。13.3 节会介绍更多关于分布式索引和一致性的内容。

## 简单性

在 SolrCloud 之前, Solr 已经为可扩展性、高可用性和一致性提供了解决方案, 对此你可能会感到惊讶。但是 Solr 不具备简单性。SolrCloud 的主要设计目标之一是简单地实现以上这几个重要的系统特性。如果实现这些目标需要手动干预来恢复, 或者日常需要复杂的安装和维护, 那么即便实现了可扩展性和容错性, 这样做的意义也不大。每个人对于“简单性”的定义都不一样, 本书认为 SolrCloud 已经从两个方面实现了此目标:

- 日常操作很简单——一旦设置好之后, 运行 Solr 集群不会比运行单个 Solr 实例复杂。
- 节点故障恢复很简单——一旦已经解决造成节点故障的问题, 可以容易地并且自动地将节点重新添加回集群。已恢复的节点可以与其所在分片的代表保持同步。本章稍后会介绍基本的系统管理任务。

SolrCloud 的确需要额外的组件, 最值得一提是 ZooKeeper, 下一节会介绍相关内容。除了初始化配置, 我们根本不需要过多担心, 因为 ZooKeeper 基本上是“黑盒”技术。当然, SolrCloud 也处在不断发展之中, 特别是在监测大型集群方面还有待进一步发展。

## 灵活性

理想的搜索系统的最后一个特性是, 具有自动扩容能力。具体来说, Solr 能够具有添加更多副本以分配查询处理的能力, 以及将大型分片分解成较小分片的能力。对于提高查询能力来说, 可以向集群添加更多的副本, 这些副本能自动与代表保持同步。

大型集群中的另一个问题是, 分片可能会溢出托管它的服务器, 这就需要将其分解成多个较小的分片。Solr 支持将一个分片分解成两个小分片。如果集群中可以成倍增加节点数量, 此功能会很有帮助。本章最后一节会谈论关于分片的问题。

## 13.2 核心概念

我们已经启动了 SolrCloud 集群, 索引了一些文档并且执行了分布式查询。另外, 我们还了解了大规模搜索引擎的理想特性以及它们如何影响 SolrCloud 的架构。接下来, 我们从集合和内核入手, 深入了解前面章节引入的一些核心概念。

### 13.2.1 集合 vs. 内核

目前为止, 我们在这本书中只接触过 Solr 的内核。概括来说, Solr 的内核是运行在 Solr 服务器中的具有唯一命名的、可管理的和可配置的索引。一台 Solr 服务器



可以托管一个或者多个内核。内核的典型用途是，区分不同模式的文档。第1章中介绍了一个房地产搜索应用示例。在此例中，因为房源信息清单和土地信息清单的模式差异很大，所以将两者放入不同的内核。

SolrCloud 引入了集合的概念。集合将具有唯一命名的、可管理的和可配置的索引扩展成不同的分片，并且分配到多台服务器上。SolrCloud 之所以需要一个新术语，因为分布式索引的每个分片都被托管在一个 Solr 的内核中，如图 13.3 所示。

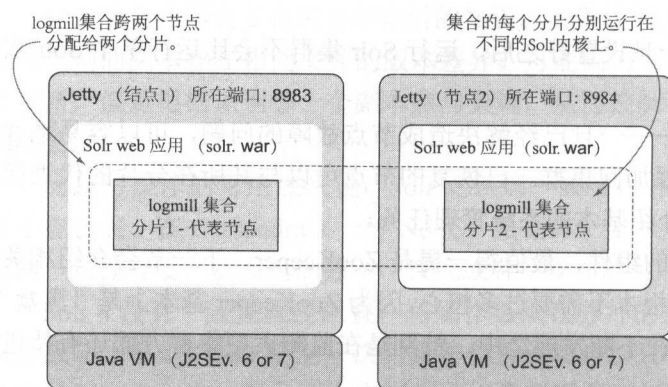


图 13.3 将 logmill 集合分配到两个独立节点的两个分片上

谈及 SolrCloud 时，更应该从分片的角度（而不是内核的角度）去思考问题。索引的分片之间是互斥关系。分片是由 Solr 内核来支撑的，这个事实仅仅是实施上的细节问题。因此，除非谈及非 SolrCloud 行为，否则本章不会使用“内核”这个表述。

正如单台 Solr 服务器能托管多个内核，一个 SolrCloud 集群也能托管多个集合。如果需要呈现不同模式的文档，可以像在单台服务器安装中使用多个内核那样，在 SolrCloud 中使用多个集合。在掌握了集合的基本概念之后，接下来了解一下 ZooKeeper。在 SolrCloud 体系中 ZooKeeper 是关键组件。

## 13.2.2 ZooKeeper

ZooKeeper 是分布式系统中的一项协调服务。Solr 将 ZooKeeper 用于三个关键操作：

- 集中化配置储存和分发
- 检测和提醒集群的状态改变
- 确定分片代表

SolrCloud 的设计者之所以选择使用 ZooKeeper，因为其成熟和稳定，很多复杂的分布式系统都在使用 ZooKeeper。本节介绍几个与 SolrCloud 有关的 ZooKeeper 核心概念。ZooKeeper 最主要的优势之一是，它为分布式系统中常见的问题提供了

强大的解决方案。ZooKeeper 社区把它们叫攻略 (recipes), 类似于面向对象的设计模式, 即: “以前遇到过此种问题, 最好使用方法 X 去解决它。” 首先我们来学习 ZooKeeper 的数据模型, 这有助于理解 Solr 如何使用 ZooKeeper 进行集群状态管理。

### ZooKeeper 数据模型

ZooKeeper 将数据组织在类似于文件系统的层次命名空间里。层级中的每一级叫 z 节点 (znode)。每个 znode 会将基本的元数据封装在里面, 例如, 创建时间、最后一次修改时间等。另外, znode 里面也能存储少量的数据。

需要注意一点, znode 不能用来存储大的数据对象。正因为这样, ZooKeeper 为每个 znode 预设了最大为 1MB 的可用存储空间。做出这样的限制是为出于性能考虑, ZooKeeper 要将 znode 放在内存中。它也可以提醒用户, ZooKeeper 不是通用的数据存储系统, 它只存储许多分布式服务器需要获得的少量元数据。ZooKeeper 的设计重点在于速度和应对许多并发请求的服务能力, 而不是要存储的数据的大小和类型的灵活性考虑。图 13.4 是 SolrCloud 使用的一些 znode。

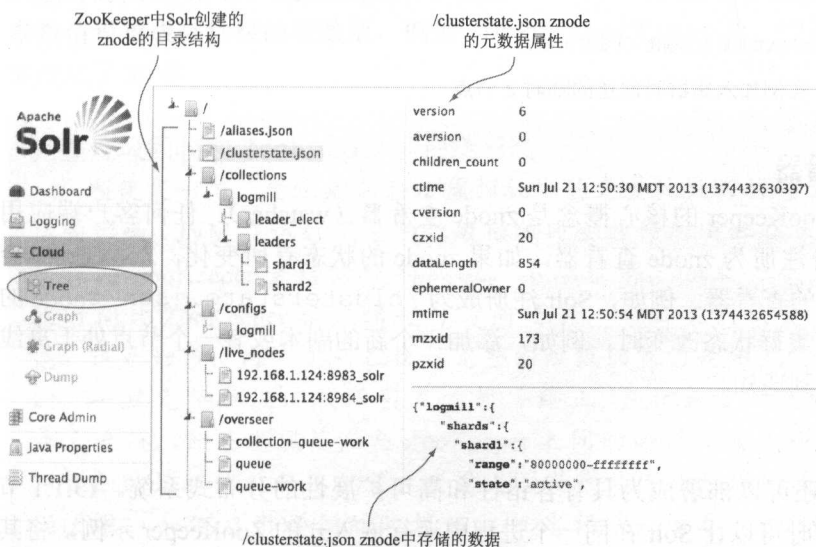
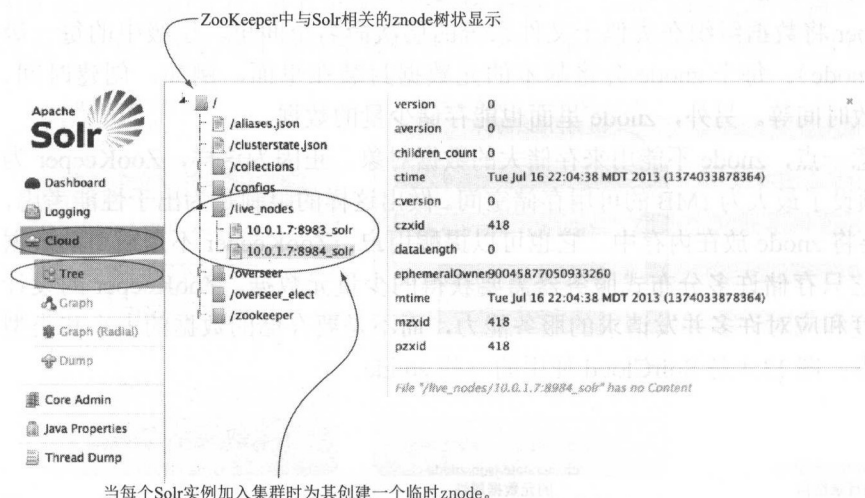


图 13.4 协调 SolrCloud 集群所需要的 z 节点的树形视图

ZooKeeper 的一个核心概念是临时 (ephemeral) znode, 它需要一个活动的客户端连接来保持 znode 一直活跃。如果创建临时 znode 的客户端应用程序离开了, ZooKeeper 会自动将临时 znode 删除。当一个 Solr 节点加入集群时, 就在 `/live_nodes` 节点下面创建临时 znode。Solr 使用 ZooKeeper 的 API 与该节点保持连接。



如果 Solr 崩溃了, 与临时 znode 的连接就自动断掉, 导致系统认为该节点也消失了。当 znode 的状态改变时, ZooKeeper 就会通知集群里的其他节点: 其中一个节点出故障了。这样 Solr 就不会尝试向故障节点发送分布式查询请求。图 13.5 中树形视图下的云面板中看到前面小节在 Solr 服务器开启的临时 znode。



当每个 Solr 实例加入集群时为其创建一个临时 znode。

图 13.5 每个 Solr 实例加入集群时创建的临时 z 节点

## znode 查看器

另外一个 ZooKeeper 的核心概念是 znode 查看器 (watcher)。任何客户端应用程序都能将自身注册为 znode 查看器。如果 znode 的状态有所变化, ZooKeeper 会通知所有已注册的查看器。例如, Solr 注册成为 /clusterstate.json znode 的查看器, 以便在集群状态改变时, 例如, 添加一个新的副本或者一个节点处于离线状态, 能够接收到通知。

## 产品配置

ZooKeeper 还可以部署成为具有容错性和高可扩展性的分布式系统。13.1.1 节介绍过, 初上手时可以让 Solr 在同一个进程中运行嵌入式的 ZooKeeper 示例, 将其作为 Solr 节点之一。然而, 在生产环境中如果这样做的话是不明智的。

对于生产环境来说, 因为 ZooKeeper 依赖于任何时候都能达到与 znode 状态一致的主要节点 (通常称为 Quorum 机制), 所以需要设置至少由三个节点组成的独立的 ZooKeeper<sup>3</sup>。如果部署三个节点, 即便少一个节点, 也不会造成停机。如果

3 “ZooKeeper: Because Coordinating Distributed Systems is a Zoo.”, Hadoop, Nov. 19, 2012, <http://zookeeper.apache.org/doc/r3.4.5/>.

ZooKeeper 脱机，Solr 也仍然能对查询做出响应，但是会拒绝接受更新。这是一种保障机制。Solr 能做出回应是因为每个节点缓存了从 ZooKeeper 接收到的集群状态。

一旦开始整体运行，就需要使用 `zkHost` 参数将连接字符串传递给 Solr。前面的嵌入式 ZooKeeper 示例中使用 `zkHost=localhost:9983`。如果准备在生产环境中运行 SolrCloud，请将此代码写入到 ZooKeeper 的连接字符串中。如果在 `zk1.example.com:2181`、`zk2.example.com:2181` 和 `zk3.example.com:2181` 上设置了由三个节点组成的集合，在生产环境中启用 Solr 时需要传递以下参数：

```
-DzkHost=zk1.example.com:2181,zk2.example.com:2181,zk3.example.com:2181
```

### ZooKeeper 客户端超时

需要理解的第二个重要参数是，ZooKeeper 客户端超时。如前所述，当有节点加入集群时，Solr 会创建临时 `znode`，以表示有活动节点。如果其中一个 Solr 节点崩溃，ZooKeeper 在超时后会自动检测到节点的崩溃。因此，用户希望超时时间尽量短，以确保处于 ZooKeeper 管理之下的集群状态能反映当前集群的状态。在 Solr 中默认的超时时间为 15 秒。通过设置 `zkClientTimeout` 参数可以改变这个值，需要将参数值设置成毫秒级的整数值，例如 `-DzkClientTimeout=30000` 会把超时时间改成了 30 秒。

### 完整垃圾回收和 ZooKeeper 客户端超时

顺便提一句，要注意在 Java 虚拟机中的完整垃圾回收对于 ZooKeeper 会话的影响。JVM 在运行完整的垃圾回收时会暂停所有正在执行的线程，包括保持 ZooKeeper 会话活跃的线程。不难理解，如果一个完整的垃圾回收时间比 ZooKeeper 客户端超时时间还长，那么实例就会显示成脱机状态。这是好事，因为实例在执行完整的垃圾回收时是不能对请求做出响应的。一旦完整垃圾回收所引发的暂停终止，Solr 会尝试重新与 ZooKeeper 建立连接。如果发现节点与 ZooKeeper 之间的会话数量不断下降，建议启用详细的垃圾回收日志记录，查看是否是因为完整垃圾回收活动耗费的时间太长。如何启用详细的垃圾回收日志记录，请参考 JVM 的文档信息。

### 集中化配置文件储存和分发

集中化配置文件管理和分发是 ZooKeeper 的强大功能之一。在 SolrCloud 集群中启用新集合时，必须要将配置文件（如 `solrconfig.xml` 和 `schema.xml`）上传到 ZooKeeper 中。当 Solr 在云模式下启动时会自动把配置文件从 ZooKeeper 中提取出来。关于这一点已经在 13.1.1 节中讲过，删除 `conf/` 目录之后，启动第二个实例。如果

需要改变 `solrconfig.xml` 文件的配置,将更新版本上传至 ZooKeeper,再重新载入集合,以便触发集群里的所有节点,将集中化存储的更新应用到节点上。

这是 SolrCloud 的一个很强大的功能,因为它可以让用户在任何时候新增节点,并且节点也能应用最新的配置文件。试想一下,要管理一个由 100 台服务器组成的 Solr 集群,需要改变 `solrconfig.xml` 文件的设置。比方说,需要在 `solrconfig.xml` 文件中,为新的搜索器监听器配置元素添加一个新的预热查询 `<listener event="newSearcher">`。此配置更新需要发送给所有的服务器,之后再重新载入所有集合。如果将此配置更新只发送到一个集中地点,让所有的服务器自动提取此更新,这岂不是更好?这是 ZooKeeper 支持 SolrCloud 的重要进程之一。

图 13.6 展示了在 ZooKeeper 中配置文件的储存位置。需要注意的是, ZooKeeper 不是用于存储大型文件的,其默认存储空间大小设置只有 1MB。

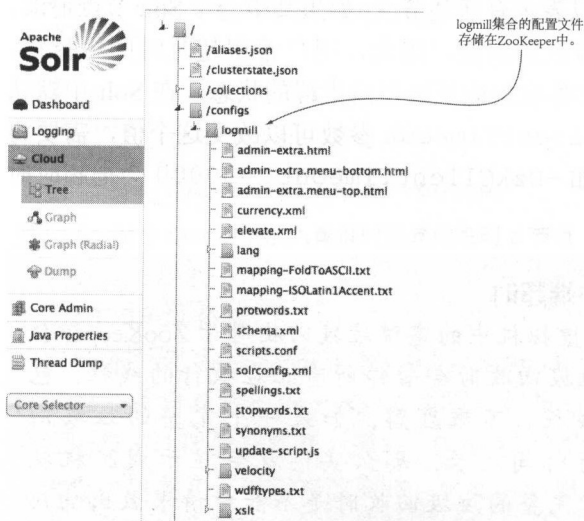


图 13.6 屏幕截图显示了 logmill 集合的配置文件在 ZooKeeper 中的存储位置

现在,我们对 ZooKeeper 的概念有了基本了解和认识,接下来讨论如何确定集合里分片和副本的合适数量。

### 13.2.3 确定分片和副本的数量

在第 12 章的 12.5 节,我们讨论过如何确定分片的合适数量,对数据进行分区,以及如何确定包含分片数据的副本的数量。基于性能和容错性等因素,例如,文件总数、文件大小、所需的索引量、查询复杂度及索引大小随时间增长的情况等,用户可能选择一个分片或上百个分片。

在 SolrCloud 配置中确定分片和副本的数量时，第 12 章讨论过的性能因素同样也适用于这里。从本质上讲，不管是否明确执行分布式搜索，在查询中加入 shards 请求参数，或者是否允许 SolrCloud 在集合中找到每个分片的所有副本，所有搜索都运行在一个或者多个 Solr 分片上。Solr Cloud 隐藏了找到每个分片位置的复杂过程。因此，只要使用 SolrCloud，而不是手工管理集群，第 12 章介绍的用于处理数据量和查询量、管理分片和复制操作的指导原则，在这里仍然适用。SolrCloud 提供了一种更简单的分布式搜索环境的管理机制，下一节会详细介绍。

### 13.2.4 集群状态管理

分布式索引和搜索依赖于各个节点才能知晓集群里其他节点的状态。例如，一个分片代表需要知道所有要接收更新信息的副本情况。任何时候都存在多种可能的节点状态，该状态决定了一个节点能否处理查询或者接收更新请求。表 13.3 介绍了 SolrCloud 中实例的几种可能状态。

表 13.3 SolrCloud 中实例状态的简要描述

状态	描述
活跃	活跃状态的节点能服务于查询和接收更新请求。活跃的副本能与分片代表保持同步。在良好状态的集群中，所有节点都处于活跃状态。
不活跃	在分片期间实例状态不活跃，则表明 Solr 实例不再参与到集合中。一旦分片活跃，已被分解的碎片就会进入此状态。
构建中	在分片期间使用“构建中”这种状态表明正在进行分片。处于此状态的分片将来自于父分片的更新请求进行缓存，但不参与查询。
恢复中	恢复实例正在运行时不能服务于查询，不过在恢复过程中它们可以接受更新请求，为的是不至于落后分片代表太多。
恢复失败	实例试图恢复但是遇到错误，即为恢复失败状态。大多数情况下，需要查看日志，手动解决实例恢复碰到的问题。
故障	实例能运行，也连接到了 ZooKeeper 上，但处于一种不能恢复的状态。例如，Solr 在进行初始化时发生故障。出了故障的实例不能参与查询和接受更新。通常故障状态都是暂时的，节点会很快转换成其他状态。
消失	实例没有连接到 ZooKeeper 上，而且有可能已经崩溃了。如果一个节点仍在运行，但是 ZooKeeper 认为它已经消失了，那么最可能的原因就是 JVM 中的 OutOfMemoryError 错误。

理想情况下，我们希望集群里的所有节点一直处于活跃状态，SolrCloud 就用于妥善处理节点故障和加入集群的新节点。在管理集群状态方面，Solr 在很大程度上依赖于 ZooKeeper。

在幕后，SolrCloud 使用监督组件在 ZooKeeper 的 /clusterstate.json 节点中储存当前集群状态的快照。每个集群只有一个监督组件，但是如果当前监督组

件出现故障，Solr 会自动选择新的监督组件。因此，监督组件不是单点故障。监督组件的选择过程类似于确定分片代表的过程。

每个 Solr 实例都会注册成 `/clusterstate.json` `znode` 查看器，以便接收状态改变的通知。例如，当新的副本加入集群时，监督组件就会更新 `/clusterstate.json` `znode`，将新节点包含进来。当 `znode` 有所改变时，ZooKeeper 会通知集群里的所有其他节点，这会触发所有节点刷新自己的集群状态缓存视图。

除了集群中处于离散状态的各个节点，其他节点可作为分片代表，也可作为副本。接下来，我们将注意力转移到如何选择分片代表，这是理解 SolrCloud 的最后一个核心概念。

### 13.2.5 确定分片代表

分片代表负责接收更新请求，并将这些更新请求协调分配给各个副本。具体来说，分片代表负责以下几项额外的更新请求，副本无法负责这些请求。

- 为分片接收更新请求。
- 在更新后的文档上增加 `_version_` 字段的值，并且强制执行乐观锁定。
- 将文档写入更新日志。
- 以并行方式将更新请求发送给所有副本并且封锁，直到收到响应。

在处理查询请求时，分片代表没有额外的工作。每个分片的主机都可以充当分片代表的角色，那么其他主机自然就是副本了。与副本一样，分片代表也参与分布式查询，这一点与主从架构不同。在主从设置里，主节点只负责索引，而从节点负责执行查询。

在了解如何确定分片代表之前，必须明白一点，不应该关注分片里的哪个节点是当前代表，也不要尝试去控制它。SolrCloud 的设计目的在于，让分片里的任何主机都能充当代表，而且能自动选择新的代表。在大多数情况下，分片代表仅仅是一个实现的细节问题，对于如何设置和操作集群并没有太大的影响。

分片代表的选择包含两方面内容：选择最初的代表和在当前代表出故障时自动选择一个新的代表。对于代表的选择需要进行集中协调，我们不希望出现，同一个分片里的两台主机都是代表的情况。你可能已经猜到了，ZooKeeper 在分片代表的确定过程中会发挥关键性作用。实际上，对于许多分布式系统来说，选择代表都是常见的需求，因此 ZooKeeper 已经将其纳入解决方案了。

为了说明代表是如何选择的，假定一个场景，一个分片有 4 个副本，它们同时加入集群，需要选择其中的一个副本作为代表。按理说，4 个节点中第一个注册的节点应该被选为分片代表。在后台，Solr 使用 ZooKeeper 的序列标志跟踪各节点各

自注册成为代表候选人的顺序。序列标志以原子方式递增，所以有多少客户端试图同时增加序列，不会造成什么影响。图 13.7 显示了 logmill 集合实例中 shard1 使用 znode 选择代表的情况。

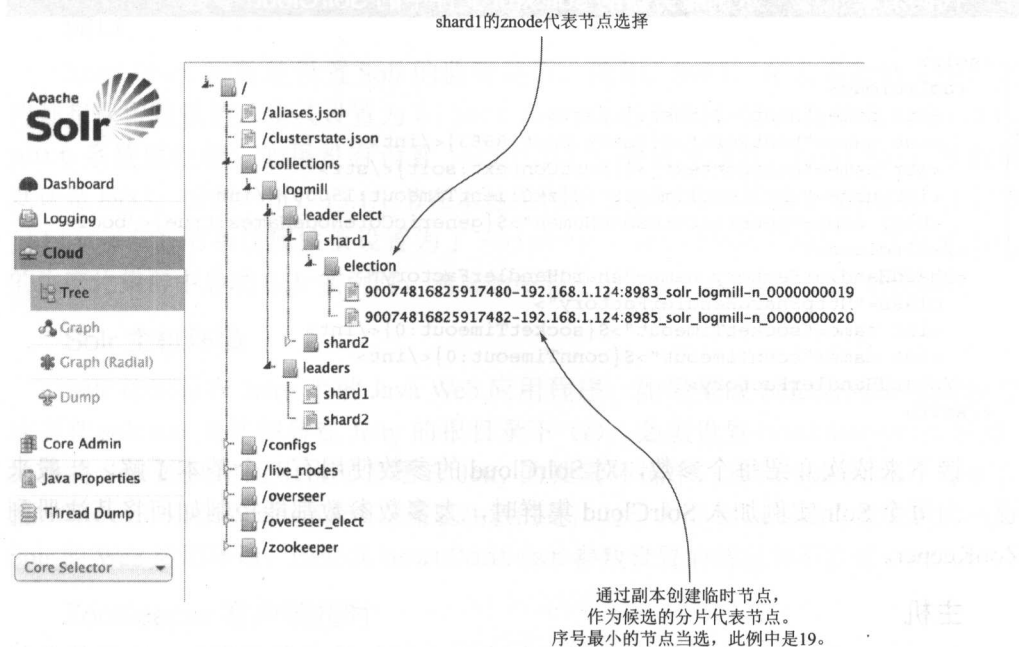


图 13.7 使用 znode 选择分片代表

简单来说，序列号最小的节点会胜出，成为分片代表。如果当前代表出故障的话，会发生什么情况呢？这种情况下需要选出新的代表，索引才能继续进行。事实证明，候选序列中下一个状态良好的节点会成为代表。如果对代表故障如何转移的细节感兴趣，推荐阅读 ZooKeeper 文档<sup>4</sup>。

### 13.2.6 SolrCloud 的重要配置

当你进行集群规划时，需要知道 SolrCloud 的一些其他配置。示例服务器自带的 solr.xml 文件默认包括 <solrCloud> 元素，在此元素中可以设置与 SolrCloud 相关的各种配置属性。学习 SolrCloud 时不必担心改变参数的问题。这里介绍它们，只是想让你知道，在 SolrCloud 用于生产环境时可以用到它们。目前可以跳过本节，当你有进一步了解需要时再细读即可。如果对 SolrCloud 的配置参数感

<sup>4</sup> “ZooKeeper Recipes and Solutions,” Hadoop, Oct. 9, 2013, [http://zookeeper.apache.org/doc/trunk/recipes.html-sc\\_leaderElection](http://zookeeper.apache.org/doc/trunk/recipes.html-sc_leaderElection).



兴趣的话,请参见代码清单 13.2,该段代码包含了示例服务器的 solr.xml 文件的 <solrcloud> 元素。

代码清单 13.2 示例服务器的 solr.xml 文件中的 SolrCloud 参数

solr.xml 中  
SolrCloud  
相关参数

```
<solr>
  <solrcloud>
    <str name="host">${host:}</str>
    <int name="hostPort">${jetty.port:8983}</int>
    <str name="hostContext">${hostContext:solr}</str>
    <int name="zkClientTimeout">${zkClientTimeout:15000}</int>
    <bool name="genericCoreNodeNames">${genericCoreNodeNames:true}</bool>
  </solrcloud>
  <shardHandlerFactory name="shardHandlerFactory"
    class="HttpShardHandlerFactory">
    <int name="socketTimeout">${socketTimeout:0}</int>
    <int name="connTimeout">${connTimeout:0}</int>
  </shardHandlerFactory>
</solr>
```

接下来依次介绍每个参数,对 SolrCloud 的参数使用有一个基本了解。一般来说,当每个 Solr 实例加入 SolrCloud 集群时,大多数参数都能控制如何将其注册到 ZooKeeper。

## 主机

ZooKeeper 必须要清楚知道集群中每个 Solr 实例的主机和端口。这些信息都存储在 /clusterstate.json znode 中,以便于每个节点都可以访问集群中的其他节点。当一个 Solr 实例加入集群时,它会对主机和端口进行注册。默认状态下,主机将被设置为 IP 地址,端口则是 Jetty 被绑定的那个端口。在图 13.1 的例子中,ZooKeeper 上注册的第一个 Solr 实例使用了 IP 地址 10.0.1.7,端口号为 8983;10.0.1.7 是本地路由器内部分发的 IP 地址。对于开发和测试,使用本地工作站的 IP 地址没有什么问题。如果在生产环境中准备部署 SolrCloud 时,建议将主机参数设置成主机名,不要使用内部 IP 地址。这样在 Solr 管理控制台中的云面板更易于理解。此外,如果将主机名分配给不同的服务器,改变 DNS 入口比在 /clusterstate.json 中更新 IP 地址要简单得多。

试想一下,如果要将其中一个 Solr 实例的主机属性设置成主机名称 solr1.example.com,我们可以编辑 solr.xml 文件,使用 <str name="host">solr1.example.com</str> 设置主机。但是,如果集群中的每个实例都需要更改的话,每更改一次就需要编辑一次 solr.xml 文件,这对于系统管理员来说是很痛苦的事情。默认情况下,solr.xml 文件将主机参数设置成 <str name="host">\${host:}</str>。值 \${host:} 是一种特殊的配置语法,允许使用 Java 的系统属性设置参数,



在冒号后指定一个默认值。这种情况下，主机属性的默认值为空。当启动 Solr 时，可以在命令行上传递 `-Dhost=solr1.example.com`，这样就可以避免每次更改集群的实例都要重新编辑 `solr.xml` 文件了。

## 端口

`hostPort` 参数能设置 Solr 的监听端口，例如，8983。在大多数情况下，用户都想保留默认值，将其设置为 `${jetty.port:8983}`，这只是使用了 `jetty.port` 系统属性值。如果没有设置 `jetty.port` 的系统属性，`hostPort` 参数值默认是 8983。回想一下，13.1.1 节在本地工作站上启动第二个 Solr 实例时，我们将 `jetty.port` 的系统属性值设置为了 `-Djetty.port=8984`。通常，在基于 Jetty 的生产化集群中启动 Solr 实例时，建议始终要传递 `jetty.port` 这个系统属性。

## Solr 主机环境

Solr 是运行在 Jetty 上的 Java Web 应用程序。如果更改 Solr 的网络应用程序或者把 `solr.war` 文件部署在 Jetty 的根目录下 (`/`)，必须设置 `hostContext` 参数。如果将 Solr 的网络应用程序部署在 Jetty 的搜索环境中，启动 Solr 时就不需要设置 `-DhostContext=search`。需要注意的是，对于大多数 Solr 的安装来说，改变 Solr 的 Web 应用环境，改变其 `hostContext` 参数设置的情况并不多见。

## ZooKeeper 客户端超时

13.2.2 节讨论过 ZooKeeper 客户端超时的一些概念。与 `solr.xml` 文件里其他参数一样，建议使用 Java 的系统属性去更改 `zkClientTimeout` 参数值。因为在整个集群里的所有节点上这个参数的值应该都是相同的。编辑 `solr.xml` 文件比使用命令行的方法更容易令人接受。

## 内核节点名

`genericCoreNodeNames` 参数应该引起注意，因为此参数有点令人困惑，对于示例 `solr.xml` 文件来说，这个参数可能没有什么作用。在大多数情况下，可以忽略此参数，将其值设置为 `true` 就行。简单地说，为集合里的内核创建唯一名称时，此参数控制 Solr 使用的命名策略。13.2.1 节谈到，一个集合分布在多个内核上。集群里的每个内核都需要有自己独特的名称。如果将参数 `genericCoreNodeNames` 设置为 `true`，Solr 会为各个内核分配一个通用名称，如 `core_node1`。如果参数值为 `false`，内核名称会包含 `host` 参数值，如 `10.0.1.7:8983_Solr_logmill`。

## 代表投票等待期

还有一个参数没有包含在示例 `solr.xml` 文件。在分片代表确定之前，这个参数

控制一个节点等待其他节点投票担任分片代表的等待时长。`leaderVoteWait` 参数为分片代表的选择提供了安全保障机制，以防止带有过期数据的节点自动成为代表，这让托管相同分片的其他节点也有机会投票成为代表。

假设有这样一个场景：托管一个分片的两个实例，其中节点 X 是代表，节点 Y 是 `shard1` 的副本。如果节点 Y 崩溃，但是节点 X 作为分片代表继续接收更新请求。在此场景中，节点 Y 已处于脱机状态，就不能与分片代表同步。如果节点 X 在节点 Y 恢复之前崩溃，那么在节点 Y 重启之后，也不应该立即担当代表角色。因为跟节点 X 相比，节点 Y 的数据是陈旧的。`leaderVoteWait` 参数的时长默认设置为三分钟，目的就是防止场景举例中的情况出现。如果节点 X 能在 `leaderVoteWait` 参数默认的时长里重新处于联机状态，则系统会恢复其原来的代表角色，节点 Y 恢复后仍然是副本角色。当然，这仅仅适用于在短时间内节点 X 和节点 Y 都同时脱机的情况。

上面介绍的安全保障机制隐含的一层是：如果节点 X 在 `leaderVoteWait` 参数默认的时间段里处于脱机状态，节点 Y 是不会恢复成活跃状态的。具体来说，节点 Y 会进入等待期，看看是否有相同分片的其他节点加入集群。这意味着，如果节点 X 不重新加入集群或者 `leaderVoteWait` 参数默认时长没有结束，`shard1` 都会保持脱机状态。只有托管一个分片的所有节点都脱机才会发生这种情况，通过每个分片使用更多的副本可以缓解这种情况。

在介入下一节之前，回顾一下本节内容要点。

- SolrCloud 使用集合这个概念来描述跨多个 Solr 实例的索引分割。从分片的角度看待问题比从内核的角度要更好，Solr 的内核仅仅是实现上的细节问题。
- ZooKeeper 是 SolrCloud 的一项关键性服务，它提供了集中化配置管理、集群状态管理和确定分片代表。
- 当定义一个集合时，必须指定分片的数量。数量的确定可以基于文件数量、文件大小、索引吞吐量、查询复杂性以及随着时间增长的索引大小。
- 监督组件会更新 `/clusterstate.json` `znode` 节点，以便集群状态有所改变时能通知其他所有节点。
- 在创建索引时，分片代表会承担一些额外工作。分片代表的确定是由 ZooKeeper 提供的方法自动选择的。
- 如果当前分片代表出了故障，系统会自动选择一个新的代表。

接下来，我们根据掌握的这些核心概念来理解分布式索引的工作原理。

### 13.3 分布式索引

从客户端的角度看，SolrCloud 里的索引文档是相同的。实际上，不需要修改索

引客户端程序就能直接使用。但是，也可以对索引客户端程序做一些潜在的优化，以提高性能和支持近实时搜索。从 Solr 服务器的角度看，索引已经发生巨大的改变。本节介绍 SolrCloud 中分布式索引的工作原理。

SolrCloud 中分布式索引的总体目标是能够将文档发送到集群的任何节点上，并且使文档能在正确的分片中被索引。此外，SolrCloud 的分布式索引目的在于，去除所有的单点故障（Single Point Of Failure, SPOF），这是达到高可用性所不可或缺的。接下来，我们从如何将文档分配给分片开始讨论分布式索引。

### 13.3.1 将文档分配给分片

当 Solr 索引新文档时，需要将其分配到一个分片上。只能将一个文档分配到每个集合的一个分片上。Solr 使用文档路由器组件来确定该文档应该被分配到哪个分片上。SolrCloud 支持两种基本的文档路由策略：`compositeID`（默认）和隐式路由。本书不讨论隐式路由，因为它会替换应用程序客户端代码的所有路由逻辑。用户可以个性化定制分片的分配，不过本节要讨论的是开箱即用的方法。本章最后一节会讨论自定义文档路由的有关内容。

正如本章前面讨论过的，必须设定分片的数量才能初始化一个集合。一旦 Solr 知道分片的数量，会给每个分片分配一个 32 位的散列值。对于 13.1 节中构建的两个分片的集群示例来说，分配给 `shard1` 的散列区间是 `80000000-ffffffff`，而 `shard2` 的散列区间是 `0-7ffffffff`（十六进制值）。系统会在每个分片之间均分这个 32 位的散列值区间。

默认的 `compositeID` 路由器会计算出文档里唯一 ID 字段的数值散列值，并将文档分配给哈希值区间包含计算得到的哈希值的分片。这表明每个发送给 SolrCloud 的文档都必须具有唯一的文档 ID 号。在两个分片的集群示例中，如果一份文档的唯一 ID 字段的散列值是 `80000001`（十六进制），那么这份文档会被分配给 `shard1`。

这就提出了一个问题，给定文档 ID 号的散列值是如何计算出来的？用于文档路由的散列函数有两个基本目标：第一，必须要快，因为在分布式 Solr 中，确定将文档分配到哪个分片上是一项基本操作；第二，应该将文档在分片之间平均分配，用户不会希望看到，分配之后其中一个分片上的文档数量是集合中其他分片的文档数量的两倍（例如偏置的散列函数）。在集群上平均分配文档，这是因为在分布式查询中，响应速度是最慢响应的那个分片为标准的。如果其中一个分片上的文档数量是其他分片的两倍，此分片有可能会成为所有分布式查询的瓶颈。

也许你对此感到好奇，其实 Solr 使用了 MurmurHash 算法<sup>5</sup>。因为该算法速度很快并且能根据散列值构造平均分配，让每个分片上的文档数量大致均衡。了解了文

---

<sup>5</sup> “MurmurHash”，<http://en.wikipedia.org/wiki/MurmurHash>。

档是如何被分配到分片上去的，接下来看在两个分片的 logmill 集合中分布式索引是如何工作的。

13.3.2 添加文档

第 5 章介绍过 Solr 有 4 种类型的更新请求：添加、更新、删除和提交。本节介绍如何将文档添加到索引中，这是理解 SolrCloud 中其他类型的更新请求工作原理的基础。图 13.8 给出了 SolrCloud 中分布式索引的过程。

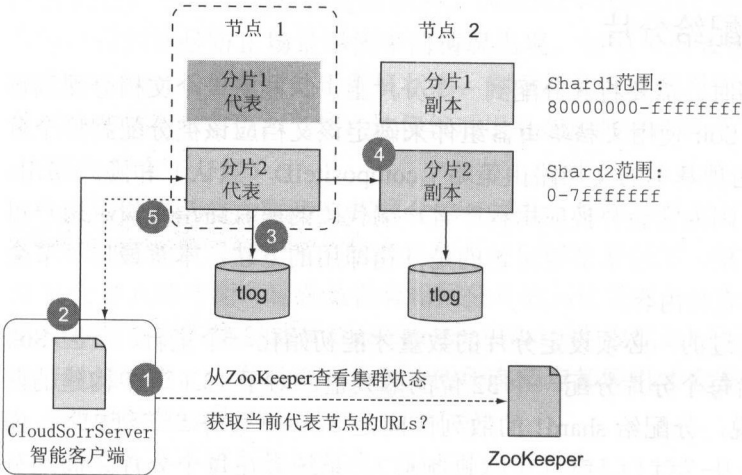


图 13.8 使用 Solr 的智能客户端，CloudSolrServer 的分布式索引过程

依次介绍图 13.8 中的索引步骤，从左下角开始。之前提到过，现有客户端应用程序不需要进行任何修改，就能将更新请求发送给集群里的任何节点，并且请求能被转发到正确的分片代表那里。然而，SolrJ 的确包括了新的 SolrServer 实现，称为 CloudSolrServer，它有助于 SolrCloud 的索引创建更加稳定有保障。以下步骤是对图 13.8 的解释说明。

步骤 1：使用 CloudSolrServer 发送更新请求

如图 13.8 所示，CloudSolrServer 连接到 ZooKeeper 上以获取集群的当前状态。这意味着，索引客户端知道集群里每一个节点的状态以及哪些节点是代表。因为 CloudSolrServer 在生成请求时会将当前集群状态信息纳入考虑范围，通常认为这是一个智能客户端。获取当前集群状态有以下两大好处。

首先，CloudSolrServer 知道哪些节点是分片代表。因为更新请求在被路由给副本之前一定会被路由给分片的代表。CloudSolrServer 通过将更新请求直接发送给分片的代表可以节省一些时间。在 CloudSolrServer 中直接使更新生效，

想象一下在客户端会发生什么。希望你猜到了 CloudSolrServer 必须实现我们在 13.3.1 节中讨论的文档路由策略。接下来将介绍其工作原理。

CloudSolrServer 的第二个好处是，它能够提供基本的负载平衡和客户端的重试逻辑。如果客户端应用程序正在索引文档时，一个节点崩溃了，CloudSolrServer 会从 ZooKeeper 那得到通知，该节点现已不可用，然后就会停止向该节点发送请求。也就是说，CloudSolrServer 会将其从负载平衡候选名单上移除。如果崩溃的节点是代表，那么 CloudSolrServer 会收到选择新分片代表的通知。因为 CloudSolrServer 在 ZooKeeper 中的 /clusterstate.json 和 /live\_nodes 的 znode 上注册了查看器，所有这些都在幕后自动发生。（请参阅 13.2.2 节中 znode 查看器的有关介绍。

13.1 节中使用的索引工具是 CloudSolrServer。其中最主要的差别在于需要具体指定 ZooKeeper 的连接地址，而不是 Solr 的 URL。参见代码清单 13.3。

代码清单 13.3 使用 ZooKeeper 地址创建 CloudSolrServer 的实例

```
String zkHost = cli.getOptionValue("zkhost", ZK_HOST);
String collectionName = cli.getOptionValue("collection", COLLECTION);
int zkClientTimeout =
    Integer.parseInt(cli.getOptionValue("zkClientTimeout", "15000"));

CloudSolrServer solr = new CloudSolrServer(zkHost);
solr.setDefaultCollection(collectionName);
solr.setZkClientTimeout(zkClientTimeout);
solr.connect();
```

使用 ZooKeeper 连接字符串连接到 SolrCloud。

指定默认的集合名称，例如，logmill。

打开 ZooKeeper 的连接，读取集群状态。

设置 ZooKeeper 客户端超时时间（参见 13.2.2）。

在 CloudSolrServer 中新增文档时，它将在集群中的所有分片代表之间平衡负载。

### 步骤 2：将文档分配给正确的分片

回到图 13.8 中的步骤 2，CloudSolrServer 需要使用文档路由进程来确定将文档发送到哪一个分片上，这曾在 13.3.1 节中提到过。例如，在图 13.8 中，文档发送给 shard2，因为 shard2 的散列区间是 0-7fffffff。一旦分片代表被选定，CloudSolrServer 会使用基于 HTTP 的 Solr API 将更新请求发送到正确的分片代表上。

从理论上讲，单个文档的添加过程非常简单。但是，如果在索引应用中将多个文档添加到 CloudSolrServer 里会发生什么呢？CloudSolrServer 需要确定这一批的每个文档应该分配给哪个正确的分片。在幕后，CloudSolrServer

将该批次分解成 S 子批次，这里的 S 是分片的数量。然后，CloudSolrServer 使用文档路由过程将每个文档分配到其中的一个子批次中去。最后，CloudSolrServer 并行将子批次发送给各个正确的分片代表，这种方式适用于 SolrCloud 中的海量索引创建。

### 步骤 3：代表分配版本 ID 号

在将文档发送给副本以前，分片代表会在本地索引文档。这是为了在将文档转送给副本以前，使用更新日志验证文档并确保文档是安全持久地存储在系统里。此外，代表会给每个新文档分配一个版本号。对于现存文档，代表会将文档的当前版本与版本号作对比，以支持第 5 章提到的乐观锁定过程。

### 步骤 4：将请求转发给副本

一旦文档通过验证并且被分配给了版本号，代表就会决定哪些副本可用，并使用多线程并行将更新请求发送给每个副本。对于任何更新请求来说，都可能会有一些副本处于脱机或者宕机状态。代表在发送文件时并不关心此种情况，因为代表能够使用恢复程序对没有发送成功的文档进行恢复，参见 13.3.4 节。有一点可能不太明显，代表会将更新请求发送给那些还处在恢复状态的副本。处于恢复状态的节点会在恢复运行过程中将更新请求写入其更新事务日志。

### 步骤 5：确认写操作成功

一旦代表收到了来自于所有活跃和恢复进程中的副本的确认，它就会将确认返回给索引客户端应用程序。只要分片里至少还有一个活跃的副本，Solr 就会持续接收更新请求。然而，这种方法更偏向于写操作的可用性，虽然有可能会以失去一致性为代价。

回想在 13.2.6 节讨论的 leaderVoteWait 参数的设置问题。恢复状态中的副本在承担代表角色以前，会有一段等待时长。这有助于降低分片里更新发生不一致的概率。然而，如果在指定时间内，先前的代表没有成功恢复，那么副本就有可能承担代表的角色，这意味着，在此种情况中一些已接受的写操作会丢失。

Solr 未来的改进计划是提供一种可选择方法，执行大多数 quorum 机制以接受 SolrCloud 中的更新<sup>6</sup>。比起写操作的可用性来说，这种可选方法更注重一致性。如果写操作不能被分片里的大部分副本所接受，那么写操作会失败。在假设的场景中，如果一个分片只有一个活跃副本，写操作就会失败，因为在只有两个副本的分片中，一个节点并不代表大多数。如果分片被复制到三个节点上，其中一个节点出了故障，那么写操作会成功，因为分片还有两个活跃节点。

---

6 SOLR-5468, “Option to enforce a majority quorum approach to accepting updates in SolrCloud,” <https://issues.apache.org/jira/browse/SOLR-5468>.



当然，在服务器端引起写操作失败只会将问题带给客户端应用程序，但是至少这能摒除代表和副本之间不一致的可能性。这个问题说明了，在构建分布式系统过程中，在写操作的可用性和一致性之间会存在权衡取舍的问题。

## 提交

除非文件被提交了，否则在搜索结果里是看不到它们的。在分布式索引中，当把提交请求发送给任何节点时，系统会将提交请求转发给集群里的所有节点，以便顺利提交每个分片。简单地说，在需要打开新的搜索器时，客户端应用程序代码应该发送硬提交请求，SolrCloud 会将提交请求传播给集群里的所有节点。

### 13.3.3 近实时搜索

如果不学习近实时搜索（NRT），那么对 SolrCloud 的了解就是不完整的，因为它是 SolrCloud 设计背后的主要驱动力之一。近实时搜索能使文件在被索引之后的数秒内就出现在搜索结果中，因此会使用到近似合格（near qualifier）。为了实现近实时搜索，Solr 提供了软提交机制，它能避免硬提交中成本高昂的操作，例如，将存储在内存的文件写入到磁盘上。

如在上一节介绍的，分片代表在对索引客户端程序做出响应之前，会将更新请求转发给所有副本。这能确保所有分片产生一致的搜索结果。这也说明，此设计决定将更新发送给所有副本在很大程度上依赖于近实时搜索支持。与此相反，主从系统不能支持近实时搜索，这是因为近实时搜索的目的是使文档在被添加到索引之后的大约一秒钟内就能出现在搜索结果中。基于主从系统的复制操作依赖于将整个片段从主节点复制到从节点。如果 Solr 必须每秒都将片段复制到从节点上，那么引擎就会建立许多小片段，查询性能将大大受影响。

让我们来看在 logmill 示例应用中如何使用近实时搜索。日志聚合是使用此类型搜索的一个很好用例，因为大多数企业都想尽快看到关键应用产生的错误。事实证明，我们可以使用任何现有客户端应用程序在近实时搜索中构建索引。只需要在 solrconfig.xml 文件中更改配置就能启用近实时搜索。代码清单 13.4 给出了启用自动软提交需要做出的更改。

#### 代码清单 13.4 在 solrconfig.xml 文件中启用自动软提交

```
<autoSoftCommit>  
  <maxTime>1000</maxTime>  
</autoSoftCommit>
```

← 每 1000 毫秒发出一个软提交。

由于软提交成本更低，所以每隔几秒就可以发起一个软提交，这样就能在近实时搜索中看到新近被索引的文件。然而，必须要记住在某个时间点仍然需要执行硬



提交，以确保文件最终被写入到永久存储里。

当执行软提交时，Solr 必须打开新的搜索器，让软提交的文档在搜索结果中可见。这意味着，Solr 还必须让所有缓存与软提交带来的改变保持一致。在软提交之后打开新的搜索器，Solr 会对缓存进行预热，执行在 `solrconfig.xml` 中配置好的预热查询。因此，缓存自动预热设置和预热查询的执行速度必须比执行软提交的速度更快。例如，如果每隔两秒钟执行一次软提交，那么预热查询和缓存自动预热的完成时间就不应超过两秒，否则会有太多的搜索器被打开，这会导致后续提交失败。此处的关键点在于，执行软提交时，需要为近实时搜索恰当地调整缓存和预热查询的配置。简单地说，应该根据缓存自动预热和预热查询的最低限度进行调整。

尽管近实时搜索是一个很强大的功能，但并不是说一定要将它与 SolrCloud 一起使用。不使用软提交，近实时搜索也是完全可以接受的。这里给出一条建议，除非真的需要让索引文档近实时出现在搜索结果中，否则不要使用它。不要觉得使用 SolrCloud 时就一定要使用近实时搜索。使用软提交的缺点之一是，缓存经常会无效。

下一小节介绍在分布式索引中出现了故障后，Solr 如何对代表和副本保持同步。

### 13.3.4 节点恢复过程

在设计和运行大型分布式系统时，节点出现故障是普遍现象，在商用硬件上扩展系统时尤其如此。当节点出现故障时，只需简单使用复制操作来保护系统，无须购置昂贵的容错硬件。此外，还需要升级 Solr，修复错误和增加新功能，或修改 JVM 设置。最坏的情况是，集群中随时都可能出现脱机节点。不过，SolrCloud 能够妥善地处理脱机节点。

SolrCloud 提供两种基本的恢复方案：对等同步（peer sync）和快照复制（snapshot replication）。根据在节点脱机状态下错过了多少更新请求（增加、删除和更新），这两个方案的恢复过程是有差异的。

- 对等同步——如果故障时间短暂，并且处于恢复状态的节点只错过了少数更新请求，那么节点会从分片代表的更新日志中获取更新请求。目前错过更新请求的数量上限硬编码为 100。如果错过的数量超过了此限制，恢复节点将从分片代表那里获取完整的索引快照。
- 快照复制——如果一节点较长时间都处于脱机状态，导致它不能与分片代表保持同步状态，那么它会使用 Solr 的基于 HTTP 的复制操作，基于索引的快照进行恢复。

大多数情况下都不需要担心这些过程，因为节点能将其索引状态与代表做比较，自动启用合适的处理程序。这也表示可以在任何时间将副本添加到集群里，因为它能自己从分片代表那里获取完整的索引。

以上介绍了 SolrCloud 中分布式索引的工作原理，接着来介绍分布式查询的工作原理。

## 13.4 分布式搜索

一旦将索引分片，就会出现新问题，必须查询所有的分片才能得到完整的结果集。查询集合里的所有分片并创建统一结果集的过程称为分布式查询。distrib 参数能确定查询是分布式的还是本地的；启用了 SolrCloud 模式后，distrib 参数默认值为 true。将 distrib 参数值设置为 false 则会禁用分布式查询，只能借助本地索引执行查询。

### 13.4.1 多阶段查询流程

分布式查询的工作方式不同于非分布式查询，这是因为 Solr 需要将各个分片的结果汇总，然后将其合并成单个结果，最后对客户端做出响应。图 13.9 给出了 Solr 使用多阶段查询流程执行分布式查询，以下详细介绍各个步骤。

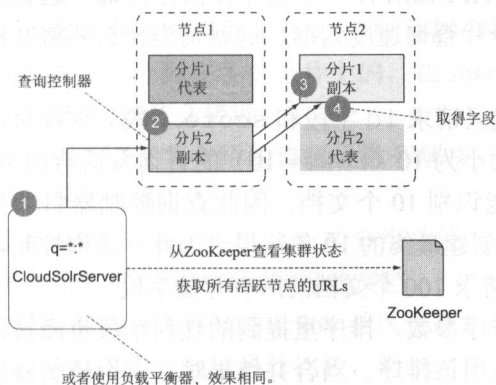


图 13.9 在 SolrCloud 中的两阶段分布式查询过程

#### 步骤 1：客户端发送查询请求至任何节点

当客户端应用程序向集群里的任意节点发送查询请求时，分布式查询从这里开始。比较常见的情况是，使用负载均衡器将查询请求分配到集群中各个不同的节点上。如果应用程序使用 SolrJ，那么 CloudSolrServer 类就会充当一个简单的分配查询请求的负载均衡器。如果没有使用 SolrJ，可以使用专门的负载均衡器或者一些像循环 DNS 地址的简单方法访问集群。

## 步骤 2：查询控制器接收请求

分布式查询会发生在多个阶段。接收初始请求的节点是查询控制器（或者聚合器），它要负责创建统一的结果集并返回给客户端。集群里的任何节点都能充当任何查询的控制器。

在 SolrCloud 中，查询控制器需要知道集群里所有节点的身份，它从 ZooKeeper 处获取此信息。但是查询控制器并不是每次接收查询请求都需要从 ZooKeeper 那里获取集群状态信息，因为那样做会成为主要的瓶颈问题。相反，如 13.2.4 节讨论的，每个节点都会注册成一个监控集群状态改变的查看器。在节点出故障的通知被 ZooKeeper 发送之前，有可能查询请求就会被发送给出了故障的节点。Solr 具有基本的容错能力，通过将请求重新发送给同一个分片上的其他主机来避免这种情况的发生。

## 步骤 3：查询阶段

查询控制器会给每个分片发送一个非分布式查询 (`distrib=false`)，以确认匹配分片里的文档。控制器能通过 ZooKeeper 提供的集群信息来确定哪些节点参与查询。查询控制器使用 SolrJ 的 API 在所有分片上并行执行查询。这意味着，分布式查询的响应速度是以最慢的分片查询速度为准，这就是为什么要将文档平均分配到所有分片上的原因了。

被发送给每个分片的查询式只请求 `id` 字段和 `score` 字段。这样可以避免过早地读取存储字段。假设将页面大小为 10 (`rows=10`) 的查询发送给由 10 个分片组成的集群。因为每个分片最多能识别 10 个文档，因此查询控制器将不得不对多达 100 个文档进行合并和排序，以创建最终的 10 条结果。因此，查询控制器会等到最后的 10 个文件被确认，而不是请求 100 个文档的所有存储字段。

如果查询能够指定自定义排序参数，排序里提到的任何字段也能被请求，那么查询控制器在合并结果时就能使用该排序。当合并结果时，查询控制器能通过合计每个分片上的总查询数量计算查到的结果总数。

在继续讨论分布式查询的第二阶段之前，有必要清楚了解使用 SolrCloud 的潜在意义。设想一下，借助 10 个分片上的集合分割，分布式查询要请求 1000 个文档（行=1000）。在幕后，查询控制器必须从每个分片上请求 1000 个文档。这意味着，查询控制器需要读取并对 10 000 个文档进行排序，才能构建最终结果集。

## 步骤 4：获取字段阶段

从第一步查询开始，一旦确定了匹配的文档，控制器会就把第二步查询发送给节点的子集，以便得到能满足请求的其他字段。如果查询式请求了 10 行，那么在获取字段阶段会请求 10 个文档。在第一步查询就知道了每个节点的所需文档（步骤 3）。

如果仅仅需要的是文档的 ID 号，就不需要获取额外字段的第二步查询了。只有需要返回包含文档的分片，才会接收到第二步的查询。例如，被确认的所有文档都来自于 shard1，那么 Solr 将第二步查询只发送给 shard1。

如果查询时没有服务器托管分片的话，会发生什么情况呢？在查询阶段，控制器将尝试查询脱机的分片。这会导致请求失败，合并阶段无法完成，这是因为 Solr 不会对于查询到的不完整结果集进行返回。通过设置 `shards.tolerant=true` 作为查询参数进行干预，表示可以接受不完整的结果。

回想一下第 4 章介绍的预热查询，思考一下是否应该对其进行分布式处理。当然，答案很简单——不应该。因为预热查询的目的是预热本地搜索器。SolrCloud 能自动将 `distrib=false` 添加到预热查询参数，以确保它们只在本地索引上执行。

### 13.4.2 分布式搜索的局限性

不幸的是，并不是所有 Solr 的查询功能都能使用分布式模式。具体来说，分布式搜索有三个主要局限需要注意。

1. 倒排文档频次 (idf) 仅基于本地索引中的词频。在计算文件权重时会用到 idf，因此在分布式查询中对文档进行排序时，会有一些偏差。默认情况下，文档是被随机分配在各个分片上的，在 shard1 中一个词项的 idf 与所有分片中一个词项的 idf 会接近。<sup>7</sup>
2. 除非使用 13.7.1 节介绍的自定义散列解决办法，否则连接 (join) 在分布式模式下不起作用。
3. 为了在 SolrCloud 中使用 Solr 的分组功能（参见第 11 章），需要使用自定义散列去分配那些会被分到同组的文档。

这些局限性都是为人熟知的，Solr 社区也在积极寻求相应的解决办法。由于当前需要对跨服务器的大量数据进行比较，特别是对分布式环境中的连接和分组提出了很大的挑战。

## 13.5 集合API

第 12 章介绍了内核管理 API 的有关知识。内核管理 API 能够以编程方式管理 Solr 的内核，参见 12.6 节。SolrCloud 提供了集合的 API，可以对分布在多个物理节点上的集合进行操作。在幕后，集合 API 可能会使用内核管理 API，在集群的单个内核上执行任务。例如，当重新载入集合，集合 API 会使用内核管理 API 将内核重

---

7 A better solution for distributed idf is being developed; 参见 <https://issues.apache.org/jira/browse/SOLR-1632>.

新加载到每台服务器上。

集合 API 支持对集合进行创建、删除和别名操作。它也支持分片，在 13.7.2 节中会进一步讨论该问题。

### 13.5.1 创建集合

本章一开始手动创建了 logmill 内核，然后在云模式下启动 Solr，将 logmill 索引分配到多个节点上。之所以这样做是想要从头开始，覆盖 SolrCloud 集群的所有过程。此外，本章的目的在于尽快上手 SolrCloud 并且运行起来，因此这里不想花太多时间来介绍集合 API。由于本地工作站已经有了一个正在运行的 SolrCloud 集群，而且在现有集群中创建新集合的过程有点麻烦，因此我们来看看如何使用集合 API 来创建集合。

本节创建一个支持客户请求的新集合。这个新集合的基本思想是，支持工程师在 support 集合中发现客户报告的问题，并将问题与 logmill 集合中的日志信息关联起来。

创建集合的棘手问题之一是，在使用集合 API 创建集合之前，需要将配置文件上传到 ZooKeeper 中。在创建集合之前，ZooKeeper 要求新集合的配置文件必须存在。或者，如果新集合可以使用与现有集合一样的配置文件，那么就不需要重新复制配置文件。这种方法对于多组织用户共享环境来说很有用。在这种环境中需要将各个用户的文件分别保存在不同集合中，它们之间共享相同的配置文件。

为了创建 support 集合，我们需要先将配置文件上传到 ZooKeeper 中。对于此方案无须修改任何配置文件，不过实际应用程序在 solrconfig.xml 和 schema.xml 文件中可能会有不同的设置。为了简化过程，只需要复制 logmill 的配置文件就可以了。

```
cd $SOLR_INSTALL/shard1/  
cp -r solr/logmill/conf /tmp/support_conf
```

上面的命令将 logmill 的 conf/ 目录递归地复制到本地工作站上的一个临时位置。现在使用一个由 Solr 提供的简单命令行工具，将配置文件上传到 ZooKeeper 中。

cloud-scripts/ 目录包含了与 Solr ZkCLI 客户端应用程序交互的命令行脚本。打开命令行窗口，从 \$SOLR\_INSTALL/shard1/cloud-scripts/ 目录下执行 zkcli.sh 脚本。在 Linux 系统中，输入以下命令：

```
cd $SOLR_INSTALL/shard1/scripts/cloud-scripts/  
./zkcli.sh
```

以上命令会显示出该实用工具的用法。为 support 集合上传配置文件，可以

使用 upconfig 命令，如代码清单 13.5 所示。

### 代码清单 13.5 使用 zkcli 实用程序将配置文件上传到 ZooKeeper

```
指定 ZooKeeper 位置。 → ./zkcli.sh -zkhost localhost:9983
                        ↳ -cmd upconfig
                        ↳ -confdir /tmp/support_conf
                        ↳ -confname support
                        ← ZooKeeper 中该配置的名称。
                        ← 上传配置文件的命令。
                        ← 待上传的配置
                          文件所在位置。
```

upconfig 命令将配置文件从本地工作站上传到 ZooKeeper。执行代码清单 13.5 之后，配置文件存储在 ZooKeeper 中的 /configs/support 目录下，如图 13.10 所示。

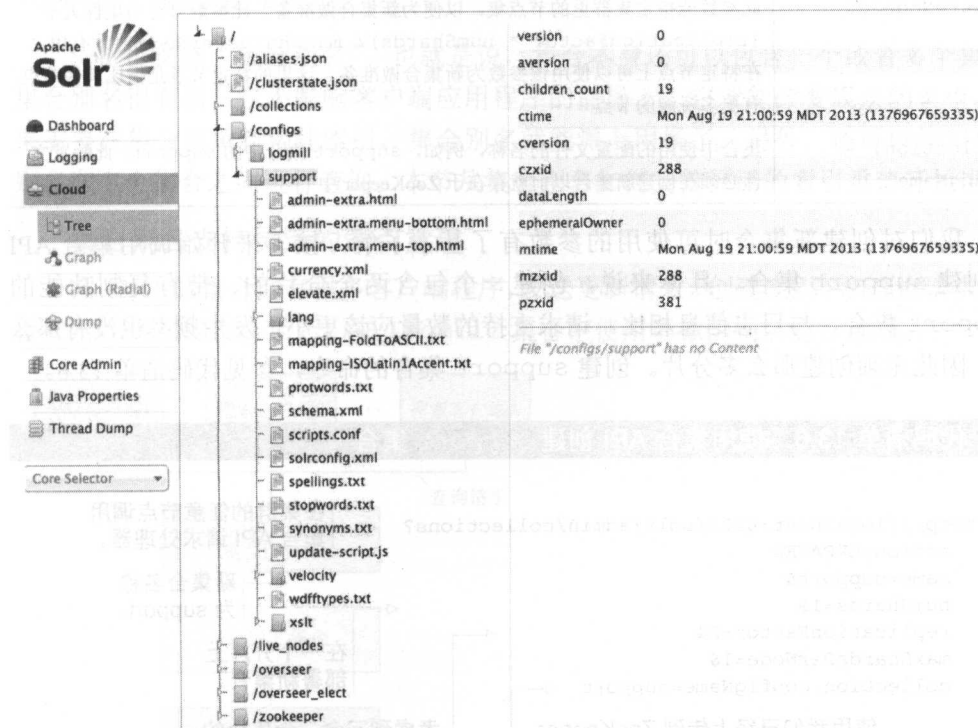


图 13.10 ZooKeeper 中 support 集合的配置文件

至此，我们已经为使用集合 API 来创建新集合做好准备了。在执行命令之前，我们回顾一下创建新集合的几种方法。表 13.4 给出了创建新集合时的可用参数。



表 13.4 用于创建新集合的参数一览表

参数	解释
name	新建集合的唯一名称，例如，support
numShards	分配集合的分片数量
replicationFactor	每个分片的副本数量，包括代表在内。设置 replicationFactor=2 会为每个分片创建一个代表和一个副本。另外，可以把这个值当成集合里文档的副本数量
maxShardsPerNode	每个物理节点上分片的最大数量。如果将此值设为 1（默认状态），则必须至少要有 numShards * replicationFactor 值一样多的节点。例如，如果设置 maxShardsPerNode=1，numShards=10 以及 replicationFactor=2，集群里必须有 20 个节点来分配新的集合，则至少需要 (replicationFactor * numShards)/maxShardsPerNode 个节点才能为新集合做准备
createNodeSet	此参数能指定集群里的节点集，以便为新集合做准备。此参数只有当集群大于 (replicationFactor * numShards)/ maxShardsPerNode 时才有用。在特定节点上可以使用该参数为新集合做准备。这里的特定节点是指比其他节点有更多资源的节点
collection.configName	集合中使用的配置文件的名称，例如，support 集合中的 support。此配置文件必须在创建新集合以前就存在于 ZooKeeper 中

我们对创建新集合时可使用的参数有了基本了解，接下来开始调用集合 API 来创建 support 集合。具体来说，创建一个包含两个分片的、带有复制功能的 support 集合。与日志信息相比，请求支持的数量应该更小，发生频率也没有那么高，因此无须创建那么多分片。创建 support 集合的命令，参见代码清单 13.6。

代码清单 13.6 使用集合 API 创建 support 集合

创建新集合的命令。

该集合仅允许每个节点一个分片。

http://localhost:8983/solr/admin/collections?

← 在集群的任意节点调用集合 API 请求处理器。

action=CREATE&  
name=support&  
numShards=1&  
replicationFactor=2&  
maxShardsPerNode=1&  
collection.configName=support

← 新集合名称为 support。

← 在一个分片上部署新集合。

← 考虑到冗余，为集合的每个文档创建两个副本。

使用我们已经上传到 ZooKeeper 的 support 配置文件。

关键在于，集合 API 可以在现有集群上定义一个新集合，并且能够控制如何在集群里的节点上进行分配。对比之前创建的 logmill 集合，让新节点依次处于在线状态。当新节点加入集群时，Solr 动态分配分片和节点角色。一般来说，创建集群之后，在已有节点上分配新集合的操作需要更多控制。集合 API 能够提供此类控制。



另外，不需要在集群里为所有节点都准备一个集合。简单地说，应根据实际需求对集合进行分片和复制。如果没有必要，就不需要将集合分配到所有节点上去。

集合 API 也支持使用 `action=DELETE` 参数来删除现有集合。删除先前创建的 `support` 集合留给读者自行操作。应当指出的是，实际上，`DELETE` 操作并不会删除磁盘上的索引数据文件，因此如果需要的话，还可以手动恢复这些文件。

在介绍集合 API 的其他用途之前，请记住一点，创建的新集合可以重用 ZooKeeper 里的现有配置文件。如果想根据日期将日志信息文件分到不同的集合里，配置都是一样的，只需要为新的日期分区添加新的集合就行。如果想根据月份将日志信息存到单独的集合里，集合 `January`、`February` 等可以共享相同的配置文件。

### 13.5.2 集合别名

Solr 也支持“集合别名”。也就是说，集合本身还可以包含一个或者多个集合。集合别名很有用，在不影响客户端应用程序的前提下，它就能改变底层的索引。如果需要在集合里重新构建索引，集合别名就能派上用场了。另外，还可以使用集合别名在多个集合之间进行查询。本节从集合别名如何支持灵活的索引重建开始讲起。

#### 集合别名支持灵活的索引重建

图 13.11 的集合拥有很多客户端程序，发送更新请求（创建索引）和查询请求（执行搜索）。如图 13.11 所示，在 `logmill` 集合中使用单独的别名进行读写。

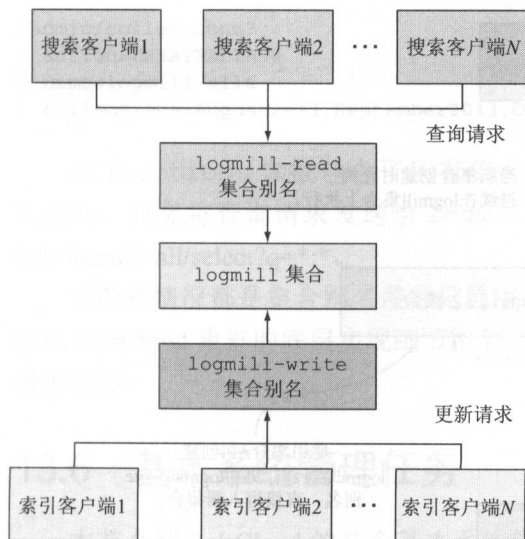


图 13.11 为 `logmill` 集合的更新与查询请求分别使用别名

稍后，我们就能看到在不影响客户端应用程序的前提下，如何对集合进行重新

索引。代码清单 13.7 是创建 logmill-write 别名的命令。

### 代码清单 13.7 创建 logmill-write 别名的命令

将别名与  
logmill 集  
合建立链  
接。

```
http://localhost:8983/solr/admin/collections?
  action=CREATEALIAS&
  name=logmill-write&
  collections=logmill
```

命名新别名。

执行 CREATE  
ALIAS 命令。

在任意节点  
调用集合  
API 请求处  
理器。

创建 logmill-read 别名留给读者自行操作。创建 logmill-write 别名以后，需要更新索引客户端应用程序，才能使用 logmill-write 环境。例如使用 `http://localhost:8983/solr/logmill-write/update`。这为索引客户端应用程序提供隔离，防止写入底层集合时需要更新。因为更改配置有可能需要重新部署，所以使用集合 API 在一个地方更新单独的别名，要比追踪所有可能给 logmill 发送更新请求的应用程序容易得多。重要的一点是，集合别名与写操作无关。集合别名存在的意义是让用户能配置索引应用程序，以向此别名发送写操作。

在图 13.11 中的索引客户端应用程序只需要与 logmill-write 别名进行交互，因此根本没有意识到 logmill 集合的存在。相反地，搜索客户端程序却只看到 logmill-read 别名的存在。这意味着能通过更改 logmill-write 别名为新集合重建索引，比如 logmill2。直到 logmill2 准备好之前，搜索客户端仍然会继续通过 logmill-read 别名查询 logmill，如图 13.12 所示。

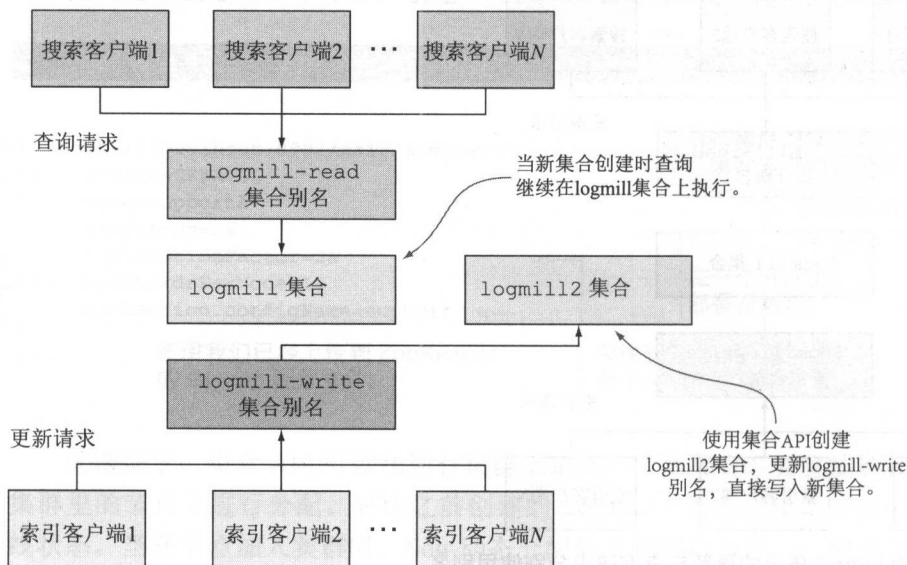


图 13.12 更新 logmill-write 别名，直接写入 logmill2 集合

要实现图 13.12 中的场景，需要对指向 logmill12 的 logmill-write 别名进行更新。使用 CREATEALIAS 命令要更新已有别名，修改 collections 参数值，如代码清单 13.8 所示：

代码清单 13.8 通过更改 collections 参数更新现有集合

```
/admin/collections?  
action=CREATEALIAS&  
name=logmill-write&  
collections=logmill12
```

更新已有别名，指向另一个集合。

### 使用集合别名进行多集合查询

集合别名的另一个常见用途是进行多集合查询，例如，支持集合里的基于日期或者时间的分区模式。假设需要根据月份对各个集合里的日志文件进行分区组织，就会产生多个集合，例如，August2013 集合、September2013 集合等。

我们可以使用两个单独的别名：logmill-recent 和 logmill-all。logmill-recent 别名包括当前和过去一个月的集合，logmill-all 包含所有月份的集合。当转到一个新月份时，使用集合 API 更新别名。具体来说，如果需要在 logmill-all 别名上新增集合 October2013，参见代码清单 13.9 中的命令。

代码清单 13.9 在 logmill-all 别名中新增集合 October2013

```
/admin/collections?  
action=CREATEALIAS&  
name=logmill-all&  
collections=August2013,September2013,October2013
```

包含别名所对应的  
所有集合。

collections 参数包含了所有集合的别名，中间以逗号分隔。要查询所有的集合，只需将查询请求发送给 logmill-all 别名，例如，[http://localhost:8983/Solr/logmill-all/select?q=\\*:\\*](http://localhost:8983/Solr/logmill-all/select?q=*:*)。

这两种情况都是集合别名最常见的用法。要注意，别名把客户端应用程序的视图从 SolrCloud 集群的底层实现细节中分离出来。下一节介绍 SolrCloud 的基本系统管理任务。

## 13.6 基本系统管理任务

本节介绍 SolrCloud 的几个基本系统管理任务。首先介绍如何将配置更改传递给集群里所有节点。

### 13.6.1 配置更新

最常见的一个系统管理任务就是更新 Solr 的配置，例如，更新 `Solrconfig.xml` 文件。在 SolrCloud 中更新配置需要两个步骤：使用 Solr 的 `zkcli` 命令行工具将更新上传至 ZooKeeper 中，然后使用集合 API 重新加载集合。举一个简单的例子，为 `logmill` 的应用程序更新缓存设置。首先，在 `$(SOLR_INSTALL)/shard1/solr/logmill/conf/solrconfig.xml` 中将过滤器缓存设置作如下修改：

```
<filterCache class="solr.FastLRUCache"
  size="60"
  initialSize="20"
  autowarmCount="20" />
```

#### 阶段 1：将更新上传至 ZooKeeper 中

使用 Solr 提供的 `zkcli` 工具将更新上传到 ZooKeeper 中：

```
cd $(SOLR_INSTALL)/shard1/cloud-scripts
./zkcli.sh -zkhost localhost:9983
  ➡ -cmd upconfig
  ➡ -confname logmill
  ➡ -confdir ../solr/logmill/conf
```

请注意，需要使用 `-zkhost` 参数指定 ZooKeeper 连接字符串。在本地的 SolrCloud 设置中通常是 `localhost:9983`，但在生产环境中，应该是以逗号分隔的 ZooKeeper 主机和端口的成对清单。`upconfig` 命令使用 `-confdir` 参数以递归方式将所有配置文件上传到已确认的目录下。

#### 阶段 2：重新加载集合

使用集合 API `http://localhost:8983/solr/admin/collections?action=RELOAD&name=logmill` 重新加载集合。

重新加载的操作会重新初始化集合里每个节点的内核，打开一个新的搜索器。因此，重新加载的代价很高，通常应该在维护时进行此操作。

### 13.6.2 滚动重启

假设我们需要更改 JVM 系统设置来调整垃圾回收行为。这需要在所有节点上重启 JVM。处理这点的最好方式就是滚动重启，这样做不会造成任何不必要的中断。具体地说，在重启期间需要逐个重启所有节点才能确保不会造成任何分片处于脱机状态。但是最好的方法应该是重启一个节点，然后等到该节点再次变成活跃状态时再继续重启下一个节点。这意味着，重启工作需要花费一些时间才能完成，但是它

能确保不用停机就能依次更新节点。当然，也能使用同样的方法去更新 Solr。该重启节点的特定操作系统细节留给读者自行探索。另外，13.6.4 节介绍的方法可以测试一个节点状态是否良好和处于活跃状态。

### 13.6.3 重启故障节点

如果其中一个节点崩溃，系统会更新节点状态，标记该节点不存在，其他节点就会停止向该节点发送更新请求。在重启该节点以前，要确保找到日志帮助诊断崩溃的原因。在解决此问题后，可以重新启动节点，它会尝试将索引与当前分片的代表保持同步。当已恢复的节点处于活跃状态时，系统会通知其他节点，则其他节点便会开始向该节点发送请求。

### 13.6.4 节点 X 处于活跃状态吗

SolrCloud 的问题之一是，使用 HTTP 确定一台服务器是否状态良好，以及能否对查询做出响应。乍一看，可能认为能尝试连接服务器端。现实并非如此，一个 ping 可能返回的假的活跃状态，这是因为它没有考虑到 ZooKeeper 中节点的状态。即使 ZooKeeper 认为节点不存在，一个 Solr 实例也可能对查询返回 200 代码（OK）。

通过在节点上使用 `distrib=false` 参数，搜索本地索引执行查询的方式来确定一个节点是否状态良好。可以从 ZooKeeper 上检索到节点的当前状态。理想情况下，我们会希望能将简单的 HTTP 请求发送给 Solr 实例，来确定该实例能对查询做出响应，以及在 ZooKeeper 中它是否处于活跃状态。Solr 已经提供了一个 ping 请求处理程序，以供检查该索引是否能对查询请求做出响应，因此这里将把它作为新增额外集群状态检查的出发点。

具体来讲，随书示例代码包含 `sia.ch13.ClusterStateAwarePingRequestHandler` 工具，它扩展了 Solr 的 `PingRequestHandler`，检查副本的集群是否为活跃状态。有兴趣的读者可以阅读源代码。修改 `solrconfig.xml` 文件中 `/admin/ping` 的请求处理程序的定义码，启用此扩展功能，参见代码清单 13.10。

代码清单 13.10 验证一个副本是否活跃的 `PingRequestHandler` 扩展

```
<requestHandler name="/admin/ping"
  class="sia.ch13.ClusterStateAwarePingRequestHandler">
  <lst name="invariants">
    <str name="q">id:0</str>
    <bool name="distrib">false</bool>
  </lst>
  ...
</requestHandler>
```

← 仅限本地，不分发 ping 查询。

← 索引之上的一个快速查询。

← 使用本书源代码中提供的扩展的完整类名。

### 13.6.5 新增副本

任何时候都可以向活跃的集群中添加更多的副本。随着这些副本上线，它们能接收到来自 ZooKeeper 基于当前集群状态的分片任务。大多数情况下，我们会想要为集群里的每个分片新增副本。使用默认的文档路由逻辑并不能让某一个分片有两个副本，同时其他分片只有一个副本，因为文档应该被平均分配到所有分片上。但是使用自定义分片时，副本数量不均衡还是有意义的，这是因为可能一个分片上的文档要多于其他分片。

要为分片新增副本，只需要启动一个新的 Solr 实例，并且传递分片的 ID，将其作为启动参数。例如，`-Dshard=shard1` 为 `shard1` 创建一个新的副本。

### 13.6.6 异地备份

复制操作能防止一些节点的偶发故障，但是如果托管集群的设备由于自然灾害或者重大故障造成整个数据中心脱机，复制操作就帮不上忙的。当然可以在不同的数据中心运行多个 Solr 集群，但是这样做成本太高，并且对于跨数据中心的分布式文档不提供支持的内置 Solr。另外一种方法是频繁地执行索引备份操作，然后将它们移动到异地，这样就能够使用另外一个数据中心的备份重建集群。如果确定要做异地备份，这里有点关于如何实现它们的建议。

随书示例代码包含了一个 SolrCloud 进行备份的小工具，参见 `sia.ch13.BackupDriver` 类。下面对 `BackupDriver` 小工具的工作原理进行分析。

- 不能直接备份索引目录，因为它可能会使备份不可用。最安全的方法是使用 Solr 的复制处理器提供的内置备份支持。如下的 GET 请求可以为 `logmill` 集合中的 `shard1` 备份索引：`http://localhost:8983/solr/logmill/replication?command=backup`。
- 需要为集合中的每个分片创建备份，所以在发布了硬提交之后，要将此命令发送到每个分片代表上。
- 实际的备份过程是在后台运行的，所以系统会立即返回备份请求。因为备份一个大型索引有可能会花上好几分钟，因此，需要一种机制来确定实际的备份过程何时完成。只有接收到备份完成的具体时间后，实用程序才调查复制处理器的细节动作。
- 一旦完成所有的备份工作，就可以将其移到异地（如 Amazon S3）。`BackupDriver` 并不提供将备份文件移动到异地的操作。

在本地工作站的 `logmill` 实例中运行实用程序，执行以下命令：

```
java -jar $SOLR_IN_ACTION/solr-in-action.jar backup
```



在运行 SolrCloud 时，需要对管理任务的类型进行规划。下一节介绍 SolrCloud 的两个高级主题。

## 13.7 高级主题

本节介绍 SolrCloud 的两个高级主题：自定义散列和分片分割。将它们视为高级主题的原因是，它们在 SolrCloud 的大多数安装中不是必须的，并不是因为它们难于理解或者使用。读完本节，在超出 SolrCloud 的正常使用时，你知道有这些功能就可以了。例如，使用自定义散列控制将文档分配到哪一个分片上。如果将来集合超出硬件存储范围，分片分割就能发挥作用了。

### 13.7.1 自定义散列

13.3.1 节提到过，默认的文档路由策略是使用唯一的 ID 字段的散列值，将文档平均分配到所有分片上。如果大多数查询都需要搜索文档的整个语料库，则适合使用这种默认策略。现在考虑这样一种情形，根据文档的一些共有属性可以将大多数查询限定在特定的文档集中。例如，多租户搜索应用程序可以在同一个索引里为不同客户托管文件。多租户环境索引下的绝大多数查询必须针对具体的租户代号。在这种情况下，根据具体的租户 ID 号在所有分片上分配文档，这不是最有效的解决方案。自定义散列可以根据一些共有字段值，如租户 ID 号，将文档分发到具体的分片上。另一种情况是根据类目分发文档。

在 logmill 示例中，假设对使用模式进行分析之后，确定大多数终端用户使用（来源 - 应用）的 ID 号进行查询，例如，solr。在这种情况下，把带有相同应用程序 ID 号的所有文档分发到相同的分片上，这样做是有意义的。当然，只把带有相同应用程序 ID 号的文件分发到所有分片上，并且在过滤查询中使用应用程序 ID 号。这种方法有两个主要优势：在很小的索引上执行大多数查询（单个分片）；可以使用分布模式下不便使用的查询功能，例如，连接和分组。

#### 复合文件的 ID 号

Solr 内置了基于一个或者多个共有字段的文档分发支持。这里需要做的是，使用特定语法创建文档 ID 号。具体来说，复合文件的 ID 号由共同字段值的前缀和感叹号 (!) 组成。要将 Solr 产生的所有日志信息发送到相同的分片上，就需要创建一个如 solr!doc123 的文档 ID 号。一般格式为 shardKey!docID，感叹号之前的值称为分片键。

在幕后，Solr 默认的 compositeId 路由器使用分片键的 16 位散列值和文档 ID 号的 16 位散列值来构造一个复合的 32 位散列值，然后根据 13.3 节讨论的标准



分片区间将其映射到正确的分片上。

不要将它与隐式路由混淆（也称为直接路由），在隐式路由中，客户端应用程序可以指定具体要索引的确切分片。通过自定义散列，可以确保带有相同分片键的所有文档可以被分配到同一个分片上，但是不能决定分到哪一个分片上。

### 在查询时间以特定分片为目标

根据共有字段值将文档分发到特定分片上，就需要查询准确的分片了。在这种情况下，客户端程序需要用 `_route_` 参数传递分片键。如果想要查询托管 solr 中日志记录的分片，需要将查询参数写为 `_route_=solr!`。另外，还可以使用逗号分隔分片键，从而查询洽谈分片。例如，`_route_=Solr!,squid!` 参数可以查询 Solr 和 squid 应用程序的分片（Squid 是一种缓存 HTTP 代理服务器）。当然，不使用 `_route_` 参数也可以查询所有集群里的分片。

### 自定义散列的局限性

使用自定义散列的最大问题是，它可能会造成集群里分片的不均衡。如果向 logmill 发送信息的某个应用生成的日志信息量是其他应用的十倍，那么托管这些信息的分片会比其他分片都大得多。出于这个原因，需要了解数据，以确定是否需要进一步细分分片键或者避免一起使用它们。

## 13.7.2 分片分割

如果需要不断地将新文档添加到 SolrCloud 中，又不删除先前索引好的任何文档，那么分片最后就会超过部署它们的硬件存储范围。当然，增长率确定了硬件存储溢出是短期问题还是长期问题。回到 logmill 示例，假设组织在过去几个月中部署了许多新的大容量应用程序，两个节点的 logmill 集群在处理新负荷中承受了过多的压力。具体来说，因为索引量太大且更新频率过快，所以索引吞吐量会逐渐放缓。此外，由于索引大小的增加，平均执行查询的时间也在增加。

我们决定在集群中新增两个节点。因为 SolrCloud 的目标之一是，处理更多的文档并保持快速的索引吞吐量，所以不能仅仅将副本添加到现有分片中，这是因为添加更多的副本只会帮助增加查询量。这里真正需要的是，将现有的两个分片分割成四个分片，从而将索引工作分配到四个节点上，而不是两个节点上。

Solr 为此情况提供了解决方案，它能允许将现有的分片分割成两个子分片。应用到每个单独分片上，分片分割过程包括以下步骤：

1. 使用集合 API 的 `SPLITSHARD` 操作将现有分片分割成两个子分片。
2. 在分片分割完成后发布硬提交，激活新的子分片。
3. 从集群中卸载原来的分片。

4. 或者，将其中的一个拆分迁移到新的服务器上。

我们将分割 `shard1` 的步骤应用到 `logmill` 的集合中。

```
curl -i -v "http://localhost:8983/solr/admin/
➡collections?action=SPLITSHARD&collection=logmill&shard=shard1"

curl -i -v "http://localhost:8983/solr/logmill/update" -H 'Content-
➡type:application/xml' --data-binary "<commit waitSearcher=\"true\"/>"
```

图 13.13 展示了 `logmill` 集合在 `shard1` 分片之前和之后的状态。<sup>8</sup> 值得注意的是：分割后的分片合在一起就等于原本的分片。这是一个重要特性，在分片操作时，我们不希望丢失复制策略。

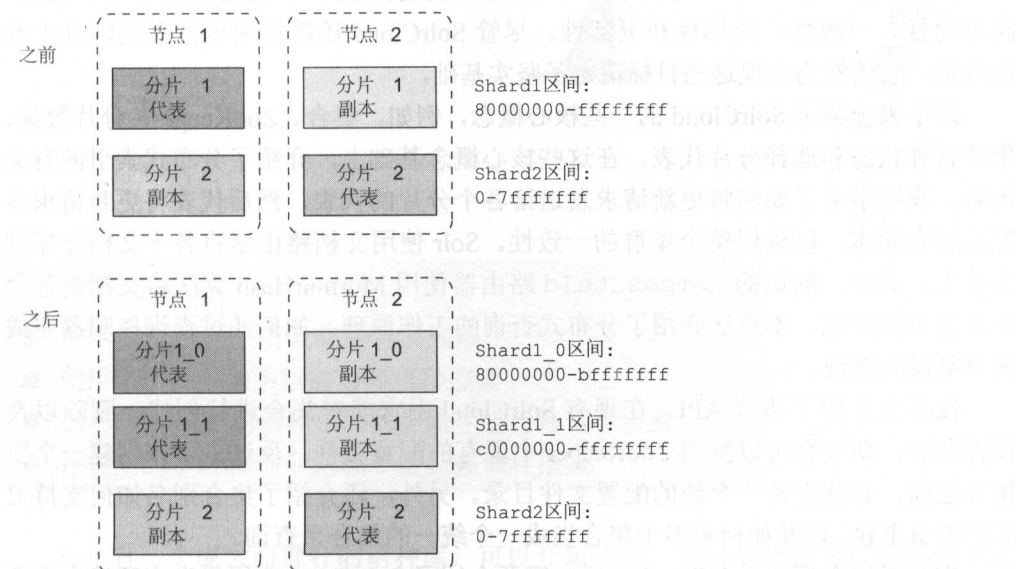


图 13.13 在 `logmill` 集合中分割 `shard1` 之前和之后的状态

假设确定需要 5 个分片来处理当前和未来的增幅，Solr 还不支持重新平衡集群添加任意数量的新节点。<sup>9</sup> 请记住，Solr 秉承迭代开发理念，当功能稳定下来就会进行发布，即使还有很多完善工作要做，例如，此处的分片分割功能。

<sup>8</sup> 希望你不会迷信十三恐惧症 (triskaidekaphobia)！参见 <http://en.wikipedia.org/wiki/Triskaidekaphobia>。

<sup>9</sup> “Implement true re-sharding for SolrCloud,” Apache Software Federation, July 10, 2013, <https://issues.apache.org/jira/browse/SOLR-5025>.

## 13.8 本章小结

通过本章的学习，希望你掌握了如何使用 SolrCloud 满足可扩展性和高可用性的搜索需求。总结一下，本章开篇在云模式下启动 Solr，对一个虚构的 logmill 日志进行聚合索引。启用云模式是通过将 Solr 连接到 ZooKeeper 实例上实现的。在这个示例中，通过 `-DzkRun` 参数使用了嵌入式实例。对于生产环境而言，应该在单独的硬件上建立一个 ZooKeeper 体系。本地的 SolrCloud 集群提供了一些实践经验，例如，将索引分成两个分片、在构建索引时将文档路由给分布式分片，以及执行分布式查询等。

启动集群以后，本章讨论了 SolrCloud 架构背后的 5 个关键因素：可扩展性、高可用性、一致性、简单性和灵活性。尽管 SolrCloud 还需在简单性和灵活性上继续改善，它仍然为实现这些目标奠定了坚实基础，

接下来介绍了 SolrCloud 的一些核心概念，例如，集合、ZooKeeper、分片数量、集群管理状态和选择分片代表。在这些核心概念基础上，介绍了分布式索引的有关内容。我们学习了如何将更新请求发送给各个分片的代表，然后代表将更新请求分配给所有副本，以确保整个集群的一致性。Solr 使用文档路由器将各个文档分配到分片上，其中，默认的 `compositeId` 路由器使用 `MurmurHash` 算法将文档在各个分片上平均分配。本章还介绍了分布式查询的工作原理，如何通过查询控制器完成两个阶段的查询。

接下来介绍了集合 API。在现有 SolrCloud 中它能对集合进行创建、删除以及设置别名。新集合可以重用 ZooKeeper 中现有的配置文件。反过来，在创建一个新集合之前，必须上传一个新的配置文件目录。另外，还介绍了集合别名如何支持灵活的索引重建，以及如何将多个集合当成一个统一的集合来查询。

本章详细介绍了 SolrCloud 之后，简要介绍了 SolrCloud 应用于生产环境中需要进行规划的系统管理流程。最后，本章介绍了 SolrCloud 的两个高级主题：自定义散列和分片分割。自定义散列是使用分片键对文档进行分类，再将文档分配到分片上。分片分割将现有分片分成两个子分片，以提高集群的容纳量。分片分割是走向真正灵活性的很好一步，但是要允许用户能够向 SolrCloud 集群中添加任意数量的节点，还需要做出许多努力。

下一章将介绍如何使用 Solr 进行多语种搜索。

# 14 多语种搜索

## 本章要点

- 使用 Solr 的词干提取和语种识别库
- 使用多个独立的字段进行多语种搜索
- 在不同的 Solr 内核和相同的字段中进行多语种搜索
- 在相同的字段和 Solr 内核中进行多语种搜索

Solr 有一个庞大而健壮的语种库，可以开箱即用，支持对全世界广泛的语种进行跨语种搜索。本章介绍了这些库的概况并说明了如何在搜索应用中高效地利用这些库。本章假定读者已经对 Solr 的 `schema.xml` 文件（详见第 5 章）以及文本分析（详见第 6 章）过程有一定的理解和掌握，如果有必要读者可以参考相关章节。

很多语种都对文本搜索带来了挑战——一些语种在搜索时要将同一个词的很多种表达形式都包含在搜索结果中，一些语种（如法语、西班牙语）包含特殊的音符，一些语种（如德语和荷兰语）包含许多由两个及以上子词组成的复合词，一些语种（如汉语、日语和韩语）的字和字之间没有空格，很难区分它们。语种分析可以针对文档中的特定语种的相关规则进行分析，从而解决由语种本身带来的复杂性问题。虽然 Solr 在其示例 `schema.xml` 文件中提供了很多预置的字段类型，但这些例子只适用于一次搜索过程中使用一种语言的情况。本章将介绍 Solr 中可用的语种库，并展示如何在混合语种的单个文档或字段中创造功能强大的多语种搜索的体验。

## 14.1 为什么语种分析很重要

试想这样一句话：“约翰到达银行了”，它可能会让人联想到一个叫约翰的人走到了一个特定的地方：一个金融机构。但是如果将文本改成“航行几个小时后，约翰抵达了银行”，这可能会让人联想到一个叫约翰的人在一艘向岸边驶来的船上。这两句话都表述了“约翰到达银行了”这一事实，但是上下文语境在正确理解这句话中扮演着关键的角色。

随着自然语言处理（Natural Language Processing, NLP）领域的发展，标准文本中的重要语境线索得以识别，包括识别未知文本的语种、判定词类、识别或近似识别词的词根、理解同义词或不重要的词，以及通过词的用法来发现词和词之间的关系。目前最好的 Web 搜索引擎都在极力推测用户查询语句中包含的查询意图。如果在 Google 搜索框中输入 `define Solr`，它可能返回“Solr is an open source enterprise search platform.”如果想利用 Google 导航到自由女神像，它可能画一张地图并提供从当前的位置去往纽约埃利斯岛的路线。Google 通过处理查询语句中的词来推测用户的查询意图，并基于最可能的意图返回搜索结果。

Solr 能够像 Google 一样，解决上述复杂的查询问题吗？答案是不能，至少 Solr 在这方面不是开箱即用的。这需要用户基于自己特定领域的专业知识来添加这些智能功能。当然可以通过开源技术在 Solr 上建立一层智能应用来实现。Apache 统一信息管理架构（Unified Information Management Architecture, UIMA）是一个优秀的开源框架，它可以使用更小的、可重用的组件来构建复杂的内容分析管道。Apache UIMA 包括很多集成的工具，可以从内容中提取知识，而 Solr 为 Apache UIMA 提供了连接器。因此，如果搜索应用中需要具备复杂的内容分析功能，它们很值得考虑。

其他聚类和数据分类技术也可以用来丰富数据，它们能带来比单一的关键词搜索丰富得多的搜索体验。其中大部分搜索功能的实现超出了本书的介绍范围，读者可以参考 Grant Ingersoll、Thomas Morton 和 Andrew Farris 的著作 *TamingText: How to Find, Organize, and Manipulate It* (Manning 出版社 2013 年出版)，书中概要地介绍了自然语言处理技术的实现，其中有一整章介绍了问答系统构建的内容，这与本书前面的一些示例类似。

对于这种类型的系统，Solr 提供了开箱即用的构建模块。它包括数十个特定语种的词干提取器、一个同义词过滤器、一个停用词过滤器和特定语种的停用词表、字符/口音归一化、查询纠错（拼写检查），以及一个语种识别器。

大多数构建模块已经在前面的章节中介绍过，但是只有正确地组合使用这些组件才能使基于 Solr 的搜索更加智能。假设用户输入关键词 `engineer` 来搜索文档，却发现所有包含复数形式 `engineers` 的文档都被忽略掉了，那将是非常可惜的。

如果没有英语词干提取器或者类似的分词过滤器将复数形式 `engineers` 末尾的“s”去掉，就会导致上述结果。

测试表明，对于常见的英语单词来讲，如果没有运行词干提取器或者其他语种分析器，搜索结果的查全率将会降低 49%（不同的语种会有所不同）以上。本书 3.3 节介绍过，查全率是衡量有多少预期文档被发现的指标。如果搜索特定语种的文档时没使用语种分析，则搜索出的文档数量将很可能被人为降低。本章将介绍如何高效利用 Solr 内置的语种处理功能来提高特定语种的查全率。首先来看一下解决词形变换问题的两种方法。

## 14.2 词干提取vs.词形还原

在第 6 章中介绍过词干提取，它是程序化地确定单词词干的过程。例如，一个英语词干提取器可能会将单词最后面类似 `s`、`es`、`ed` 和 `ing` 等词尾去掉。类似地，一个英语词干器将会准确地将 `engineers` 词干化为 `engineer`（去掉 `s`），但是同时它也会将其他词如 `nurses`、`nursing` 或者 `nurse` 统一词干化为伪单词，如 `nurs`。在实践中，只要一个词的所有变化形式最后都能词干化为一个相同的词（即使它是伪单词），就是可以的，因为 Solr 在索引和查询时都会进行词干提取。这意味着用户搜索某个单词的任一变化形式都将与它所有的变化形式匹配。

但是，词干提取会出现以下两个问题。第一个问题是，如果一个单词的所有变化形式没有映射到相同的词干，搜索会出现失误匹配。使用常见的英语 Porter（波特）词干提取器，单词 `mice` 被词干化为 `mice`，单词 `mouse` 则被词干化为 `mous`。类似地，单词 `dry` 被词干化为 `dry`，而单词 `dries` 会被词干化为 `dri`。在这种情况下单词 `dry` 与 `dries` 将不会匹配，因为它们被映射到了不同的词干。

词干提取器带来的第二个问题是它经常会导致过度匹配。考虑下列单词：`generally`、`generation`、`generations`、`generative`、`generous`、`generator`、`general` 和 `generals` 英语 Porter（波特）词干提取器将前面的每一个单词词干化为 `gener`，然而这些单词并不代表相同的意思。同样的，它会将单词 `animal` 和 `animated` 都词干化为 `anim`，将含义为“钢铁”的名词 `iron` 和含义为“讽刺的”的形容词 `ironic` 都词干化为 `iron`。过度词干化产生的结果是很多单词会被错误地匹配其他不相关的单词，从而降低搜索的查准率。词干提取器运行时会得到足够多的搜索结果，但也可能得到很多不相关的、甚至错误的搜索结果。解决这个问题的一种方法是使用一种贪婪性适度的词干提取算法，另一种方法是使用更智能的基于单词的识别方法：词形还原。

词形还原是确定一个单词的词根形式的过程。词干提取是通过算法发现一个单词的词根形式，词形还原主要利用词典查找一个单词的词根。词形还原会将单词



went 还原为 go, 将单词 am、is、are、was、were、being 及 been 都还原为 be。在之前的例子中, 词形还原会将 engineers 还原为 engineer, 将 mice 还原为 mouse 以及将 nurses 还原为 nurse。图 14.1 展示了一些示例单词进行词干提取和词形还原的区别。

原词:

goose	geese	generations	generals	several	severity
-------	-------	-------------	----------	---------	----------

词干提取的词:

goos	gees	gener	gener	sever	sever
------	------	-------	-------	-------	-------

词性还原的词:

goose	goose	generation	general	several	severe
-------	-------	------------	---------	---------	--------

图 14.1 示例单词进行词干提取和词形还原的输出对比。词形还原的结果通常看起来更为精确, 因为它基于真正的词典还原词形而不是程序化地移除或替换字母

图 14.1 中的词干提取算法并没有将 goose 和 geese 词干化为同一词干, 而将一些不相关的单词例如 several 和 severity 词干化为了同一词干。相反地, 词形还原则可以查找词的原始形式并根据词典将其还原为一般形式。很显然, 词形还原比词干提取更精确, 但找到好的词典要比使用它本身更难。

其他技术也可以用于为单词寻找合适的词根。形态分析 (morphological analysis) 工具使用统计自然语言处理 (NLP) 技术, 从大型的文本库中分析得出一种语言的语种结构, 也可以达到很好的效果。然而, 目前只能通过商业供应商来获取词形还原和形态学中经过加工后有序的语种模型库。查准率和查全率之间的权衡 (参考在 3.3 小节中的讨论) 应该是决定在搜索应用中采用词干提取还是词形还原的一个重要的考虑因素。如果不运行词干提取, 搜索的查准率会很高, 所有返回的搜索结果都与搜索的关键词精确匹配; 如果运行词干提取算法就可以得到比较高的查全率, 但是不正确的匹配带来了低查准率。如果对查全率和查准率要求都比较高, 就需要向商业供应商获取词形还原和形态学的语种模型库, 一些公司已经将标准的产品与 Solr 集成。Solr 内置的词干提取器很值得尝试, 它们能很好地适用于很多开箱即用的用例。下一节将演示如何在 Solr 中使用一些常见的英语词干提取器。

### 14.3 词干提取实战

在 6.3.4 小节中, 我们已经通过 PorterStemFilterFactory 和 KStemFilterFactory 调用了两个英语词干提取器的实例, 并初步介绍了词干提取这一概念。在深入研究 Solr 支持的其他语种之前, 本节将再介绍一些英语词干提取的示例。



运行这个示例，需要先在 schema.xml 文件中添加三个新的字段类型，如代码清单 14.1 所示。

代码清单 14.1 配置英语 fieldType 的示例

```
<fieldType name="text_en_porter" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory"
      ignoreCase="true"
      words="lang/stopwords_en.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory"
      protected="protwords.txt"/>
    <filter class="solr.PorterStemFilterFactory"/>
  </analyzer>
</fieldType>

<fieldType name="text_en_minimalstem" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory"
      ignoreCase="true"
      words="lang/stopwords_en.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory"
      protected="protwords.txt"/>
    <filter class="solr.EnglishMinimalStemFilterFactory"/>
  </analyzer>
</fieldType>

<fieldType name="text_en_kstem" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory"
      ignoreCase="true"
      words="lang/stopwords_en.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory"
      protected="protwords.txt"/>
    <filter class="solr.KStemFilterFactory"/>
  </analyzer>
</fieldType>
```

在 `schema.xml` 文件中加入这三个字段类型后，在 Solr 的管理界面中就可以很容易使用分析页面来测试它们。为了让测试更容易，我们在附带的源代码 / `example-docs/ch14/english-examples/schema.xml` 路径下包含了一个示例 `schema.xml` 文件。

### 运行本章中预置的示例

在本章中，每个示例的配置文件都已被预置在不同的 Solr 内核中了，只需启动一次 Solr，便可轻松浏览所有章节中的示例。为了保证 Solr 可以成功启动并浏览章节中的所有示例，请运行以下代码：

```
cd $SOLR_INSTALL/example/  
cp -r $SOLR_IN_ACTION/example-docs/ch14/cores/ solr/  
cp $SOLR_IN_ACTION/solr-in-action.jar solr/lib/  
java -jar start.jar
```

运行 Solr 并应用它提供的配置，然后访问网址 `http://localhost:8983/solr/#/english-examples/analysis` 进入分析页面。在这个页面上，可以比较任何字段类型的分析处理结果，在本示例中，可以选择 `text_en_porter`、`text_en_minimalstem` 或 `text_en_kstem` 字段类型。图 14.2 中演示了如何使用分析页面来测试这些字段类型的输出。（在关闭详细输出模式的情况下）。

如图 14.2 所示，`text_en_porter` 字段使得词 `engineering` 被词干化为 `engin`。这是否合理呢？如果想用 `engineer` 来匹配文档中包含的词 `engine`、`engines`、`engineer` 和 `engineers`，那么它是很合理的，因为这些词都会被词干化为相同的形式 `engin`。如果觉得这种方式贪婪性太强，可以考虑使用一种贪婪性较弱的词干提取器（选择 `EnglishMinimalStemFilterFactory` 或 `KStemFilterFactory`）。如果使用过这些词干提取器，会发现它们不像图 14.3 中所示的那么具有贪婪性。

图 14.2 展示了 `PorterStemFilterFactory` 是如何将 `engineering` 词干化为 `engin`，图 14.3 展示了 `EnglishMinimalStemFilterFactory` 和 `KStemFilterFactory` 等贪婪性较弱的词干提取结果，`engineering` 仍然保持它原来的形式。虽然它们在这个特定的例子中的输出结果是一样的，但就取词干提取的贪婪性而言，`KStem` 算法往往介于 `Porter` 算法和 `English Minimal Stem` 算法之间。为了更好地理解这三个词干提取器之间的区别，表 14.1 列出了对三个词干提取器一些常见的英语单词的提取结果对比。

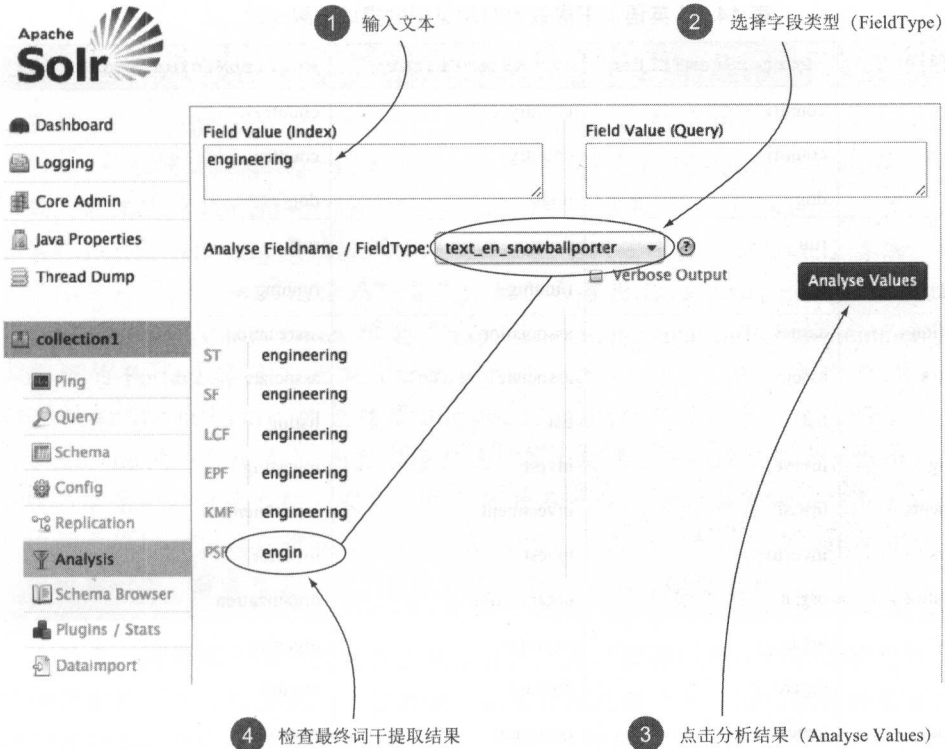


图 14.2 PorterStemFilterFactory 将 engineering 词干化为 engin

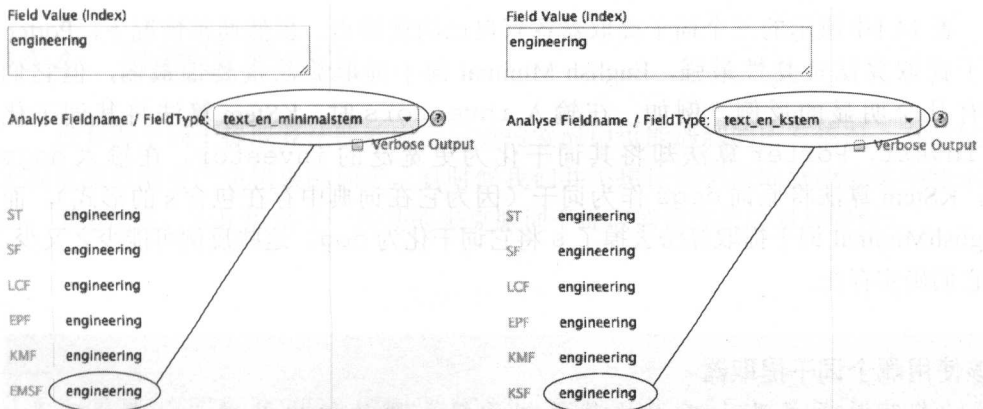


图 14.3 在 Solr 管理界面使用分析页面查看 English Minimal Stem 和 KStem 的输出结果

表 14.1 英语词干提取器对常见词的提取结果对比

原词	PorterStemFilter	KStemFilter	EnglishMinimalStemFilter
country	countri	country	country
countries	countri	country	country
dogs	dog	dogs	dog
runs	run	runs	run
running	run	running	running
association	associ	association	association
associates	associ	associate	associate
listing	list	list	listing
investing	invest	invest	investing
investments	invest	investment	investment
investors	investor	invest	investor
organizations	organ	organization	organization
organize	organ	organize	organize
organic	organ	organic	organic
generous	gener	generous	generous
generals	gener	general	general
generalizations	gener	generalization	generalization

表 14.1 中所示的三个词干提取器各有自己的优缺点。虽然通常情况下，Porter 词干提取算法贪婪性最强，English Minimal 词干提取算法贪婪性最弱，但它们仍有几个明显的反例。例如，在输入 investors 时，KStem 算法将其词干化为 invest，Porter 算法却将其词干化为更宽泛的 investor。在输入 dogs 时，KStem 算法将原词 dogs 作为词干（因为它在词典中存在包含 s 的形式），而 EnglishMinimal 词干提取算法去掉了 s 将它词干化为 dog。这些反例可能少之又少，但它们确实存在。

该使用哪个词干提取器

本节展示了几个不同的英语词干提取器的输出结果，它们分别是 PorterStemFilter、KStemFilter 和 EnglishMinimalStemFilter。同时，Solr 也包含一个独立的词干提取器——SnowballPorterFilter，它包含许多不同的词干提取操作，可以通过 language 参数进行配置（详见

14.5 小节)。对于 SnowballPorterFilter，如果将 language 的值设定为 Lovins，将会得到以往发布 English stemmer 的版本中的第一个版本<sup>1</sup>。如果将 language 设定为 Porter，将得到最早发布的 Porter stemmer 版本<sup>2</sup>。如果将 language 设定为 English，将会得到 Porter 算法的最近修订版，常被称作 Porter2<sup>3</sup>。PorterStemFilter 实现的 Porter stemmer 算法是原始的 Porter stemming 算法轻微修改版，但是它仍被认为不如 Porter2 算法。那么实际应用中应该如何选择呢？首先，如果准备用 SnowballPorterFilter，通常会使用 Martin Porter 推荐 (<http://tartarus.org/~martin/Porter-Stemmer/>) 的最新的 Porter2 算法 (此时 language="English")。但是在速度测试中，PorterStemFilter 的速度是 SnowballPorterFilter 速度的两倍 (归功于 Lucene 的实现细节)，而且它经常被推荐为处理英语的默认选择，因为词干提取的质量差异对绝大部分搜索应用而言并不重要。所以如果需要一个贪婪性强的词干提取器，应该选择 SnowballPorterFilter 和 PorterStemFilter 中的某一个，前者改进了词干提取算法，而后者拥有速度上的优势。

总之，没有哪个词干提取算法是完美的，所以需要选择一个能够满足绝大多数实例需求的词干提取算法。一般来说，如果更在意查全率，则应该选择贪婪性强的词干提取器；相反，如果更在意查准率，则应该选择一个贪婪性较弱的词干提取器。当词干提取让人感到困惑时，还可以用 Solr 提供的实用工具来帮助理清思路，这将会在下一节介绍。

## 14.4 处理边界情况

没有哪个词干提取算法能够完美应对搜索应用可能遇到的所有情况。有时候词干提取器可能会产生错误的词干，有时候我们并不想让词干提取器改变某些特定的词。幸运的是，Solr 提供了一些功能来克服词干提取器实现的自身局限性。

- 
- 1 The Lovins stemming algorithm, including links to resources, <http://snowball.tartarus.org/algorithms/lovins/stemmer.html>.
  - 2 The Porter stemming algorithm, including links to resources, <http://snowball.tartarus.org/algorithms/porter/stemmer.html>.
  - 3 The English (Porter2) stemming algorithm, including links to resources, <http://snowball.tartarus.org/algorithms/english/stemmer.html>.

### 14.4.1 KeywordMarkerFilterFactory

有时候我们并不希望应用程序对某些特定的词进行词干提取，这些词通常属于专有名词（如 Manning Publications 中，单词 Manning 可能会被词干化为 man），但也可能是开发者指定的任何单词。KeywordMarkerFilterFactory 允许开发者指定一个受保护词表，其中的单词会被 solr 中所有的词干提取器忽略。开发者只需在分析链之前添加 KeywordMarkerFilterFactory，并指定一个包含受保护词表的文件，就能避免 14.2 节中提到的词干提取器带来的两大问题。在之前的英语词干提取示例中包含了 KeywordMarkerFilterFactory，例如推荐的 Kstem 配置如下：

#### **schema.xml**

```
<fieldType name="text_en_kstem" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory"
      ignoreCase="true"
      words="lang/stopwords_en.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory"
      protected="protwords.txt"/>
    <filter class="solr.KStemFilterFactory"/>
  </analyzer>
</fieldType>
```

#### **protwords.txt**

```
listing
manning
```

在这个示例中，protwords.txt 文件中包含了需要被词干提取器 KStemFilterFactory 忽略的所有受保护的词（listing 和 manning）。开发者可以根据实际需求制定受保护词表。

### 14.4.2 StemmerOverrideFilterFactory

有时候，仅仅通过 KeywordMarkerFilterFactory 来保护单词不被错误地提取词干是不够的。如果想修改有问题的词干提取结果，StemmerOverrideFilterFactory 是不错的选择，它使用一个自定义词干提取词典，而且是先于后续词干提取器运行的，示例如下：

#### **schema.xml**

```
<fieldType name="text_en_kstem_with_overrides" class="solr.TextField"
  positionIncrementGap="100">
```



```

<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.StopFilterFactory"
    ignoreCase="true"
    words="lang/stopwords_en.txt"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.EnglishPossessiveFilterFactory"/>
  <filter class="solr.KeywordMarkerFilterFactory"
    protected="protwords.txt"/>
  <filter class="solr.StemmerOverrideFilterFactory"
    dictionary="stemdict_en.txt" />
  <filter class="solr.KStemFilterFactory"/>
</analyzer>
</fieldType>

```

#### stemdict\_en.txt

```

dogs      dog
runs      run
investors investor

```

不管选择哪种词干提取算法, 在分析链中将 KeywordMarkerFilterFactory 和 StemmerOverrideFilterFactory 结合起来使用, 就可以得到所有保护单词和修改错误词干化结果的主要工具了。到目前为止, 本章已经演示了几种英语词干提取器, 如 Porter Stem、EnglishMinimal Stem 和 KStem filters, 还有通过 SnowballPorterFilter 可以使用的英语词干提取器, 如 Lovins、原版 Porter 和 English (Porter2) 词干提取器。目前我们只是以英语为例来说明了 Solr 的功能, 不过它只是 Solr 支持的多语种处理功能的冰山一角。

## 14.5 Solr支持的语种库

本章以英语为例展示了基本的词干提取操作, 不过 Solr 还支持很多其他的语种。虽然为每个语种都展示一个输入/输出的示例是不现实的(读者可以在 Solr 的管理界面通过分析页面进行操作), 不过本章还是列出了 Solr 中所有的词干提取器, 以供读者自行探索。

### 14.5.1 特定语种的分析器

由于不同语种之间的巨大差异, 在定义 Solr 中的字段以正确处理某一语种时, 并没有一个统一的模型可供遵循。一些语种需要用它们自己的词干过滤器, 其他语种则需要多种过滤器来处理不同的语言特征(如字符归一化、重音消除, 甚至需要自定义小写转换), 一些语种由于语言分析的复杂性需要使用它们独有的分词器。表 14.2 展示了特定语种分析器之间的差异, 其中常用语种组件已用粗体标出。

如表 14.2 所示, Solr 本地就支持大量语种的特定语种分析, 不同语种的配置复



杂性相差很大。更多语种配置的示例见附录 B。如果想将任意语种添加到应用程序的模式中，表 14.2 中列出的绝大多数语种的分词器和过滤器的组合都能在 Solr 的示例 schema.xml 文件中找到对应的字段类型。

表 14.2 语种分析器链之间的差异

语种	示例分析器链
阿拉伯语	StandardTokenizerFactory LowerCaseFilterFactory StopFilterFactory [words="lang/stopwords_ar.txt"] <b>ArabicNormalizationFilterFactory</b> <b>ArabicStemFilterFactory</b>
CJK（汉语、日语、韩语）	StandardTokenizerFactory <b>CJKWidthFilterFactory</b> LowerCaseFilterFactory <b>CJKBigramFilterFactory</b>
希腊语	StandardTokenizerFactory <b>GreekLowerCaseFilterFactory</b> StopFilterFactory [words="lang/stopwords_el.txt"] <b>GreekStemFilterFactory</b>
印地语	StandardTokenizerFactory LowerCaseFilterFactory <b>IndicNormalizationFilterFactory</b> <b>HindiNormalizationFilterFactory</b> StopFilterFactory [words="lang/stopwords_hi.txt"] <b>HindiStemFilterFactory</b>
日语	<b>JapaneseTokenizerFactory</b> [userDictionary="lang/userdict_ja.txt"] <b>JapaneseBaseFormFilterFactory</b> <b>JapanesePartOfSpeechStopFilterFactory</b> [tags="lang/stoptags_ja.txt"] <b>CJKWidthFilterFactory</b> StopFilterFactory [words="lang/stopwords_ja.txt"] <b>JapaneseKatakanaStemFilterFactory</b> LowerCaseFilterFactory
波斯语	PersianCharFilterFactory StandardTokenizerFactory LowerCaseFilterFactory <b>ArabicNormalizationFilterFactory</b> <b>PersianNormalizationFilterFactory</b> StopFilterFactory [words="lang/stopwords_fa.txt"]

续表

语种	示例分析器链
罗马尼亚语	StandardTokenizerFactory LowerCaseFilterFactory StopFilterFactory [words="lang/stopwords_ro.txt"] SnowballPorterFilterFactory [language="Romanian"]

虽然本章不能详细介绍每种语言的 xml 配置,但是有一些特殊的配置值得提及。首先,来看一下最简单的示例——罗马尼亚语的推荐配置:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.StopFilterFactory" ignoreCase="true"
    words="lang/stopwords_ro.txt"/>
  <filter class="solr.SnowballPorterFilterFactory"
    language="Romanian"/>
</analyzer>
```

这条分析器链——StandardTokenizerFactory、LowerCaseFilterFactory、StopFilterFactory 和 SnowballPorterFilterFactory——为很多语种建立了基本的结构模型。绝大多数语种都可以使用 StandardTokenizerFactory,其后跟随某些小写转换处理。一些语种有自己的小写过滤器来处理特定语种的小写转换(如 Turkish-LowerCaseFilterFactory、IrishLowerCaseFilterFactory 和 GreekLowerCaseFilterFactory),但绝大多数语种都使用标准的小写过滤器——LowerCaseFilterFactory。Solr 还为很多语种提供自定义的停用词文件,可以和 StopFilterFactory 一起使用。

从表 14.2 中可以看出,一些语言如阿拉伯语、波兰语和印度语等,需要特殊的字符归一化过滤器。他其我们看到还有几种过滤器在各语种的分析器中会用到,但是前面提到的过滤器照顾到了绝大多数特定语种的特点,通常会提高这些语种的搜索质量。

大多数分析器链的最后一个特征是词干提取器的使用。绝大多数语种都有一个或更多可使用的特定语种词干提取器。在有多于两个词干提取器的语种中,可以根据词干提取的贪婪程度选择合适的词干提取器。记住,使用贪婪性越高的词干提取器,查全率会越高(更多预期文档将被检出),但查准率越低(你会检索到很多不期望得到的文档)。究竟哪种词干提取器更适合自己的搜索应用,这完全取决于自身的判断。

一般来说,从词干提取器的名字能看出它提取词干的贪婪程度。MinimalStem 过滤器提取词干的贪婪性一般是最低的,接下来依次是 LightStem 过滤器、StemFilter、SnowballPorter 过滤器。表 14.3 分别提供了它们的示例。

表 14.3 贪婪性由低到高的词干提取器的对比

最不强势	较不强势	较强势	最强势
PortugueseMinimalStem*	PortugueseLightStem*	PortugueseStem*	SnowballPorter*
FrenchMinimalStem*	FrenchLightStem*		SnowballPorter*
GermanMinimalStem*	GermanLightStem*		SnowballPorter*
NorwegianMinimalStem*	NorwegianLightStem*		SnowballPorter*
GalicianMinimalStem*		GalicianStem*	
	SpanishLightStem*		SnowballPorter*
	HungarianLightStem*		SnowballPorter*
	ItalianLightStem*		SnowballPorter*
	RussianLightStem*		SnowballPorter*
	SwedishLightStem*		SnowballPorter*
		ArabicStem*	
		BulgarianStem*	
		CzechStem*	
		GreekStem*	
		HindiStem*	
		IndonesianStem*	
		JapaneseKatakanaStem*	
		LatvianStem*	

在 schema.xml 文件中，每个词干提取器的全称都以 FilterFactory 结尾，例如 PortugueseMinimalStemFilterFactory。这个词干提取器内部为词干提取创建 TokenFilter。

从表 14.3 中可以看出的主要一点是，Solr 对很多语种都提供了开箱即用的具有不同查全率和查准率的词干提取器。SnowballPorter 词干提取器支持很多语种，通常作为默认配置，尽管它被认为是贪婪性最强的词干提取器。如果想要达到最佳的搜索体验，不妨在应用实例中多测试几种与应用实例支持的语种相适应的词干提取器，比较一下它们的效果。虽然本章不能将这些词干提取器如何处理各特定语种的细节一一详述，不过目前介绍的内容应该已经能说明 Solr 中词干提取器库的常用性，以及如何测试和使用它们。

14.5.2 基于词典的词干提取（Hunspell）

通过调用 HunspellStemFilterFactory，Solr 还支持另一个词干提取器。正如 SnowballPorterFilter 支持很多语种一样，Hunspell 支持许多词干提取算法。事实上，Hunspell 词干提取器目前支持 101 种语言，因此，如果发现它不支持你的应用程序锁需要的语言，那么是有必要的再次检查一下。它是一个基于词典的词干提取器，这就意味着它只对包含在词典中的词有作用。因此，Hunspell 词干提

取器需要为每个语种下载自定义文件（名称为 dictionary and affix files），配置如下（此示例针对斯洛伐克语）：

```
<filter class="solr.HunspellStemFilterFactory"
  dictionary="sk_SK.dic"
  affix="sk_SK.aff"
  ignoreCase="true" />
```

因为 Hunspell 词干提取器是基于词典的，所以它对不同语种处理的质量差别取决于各语种公开的 dictionary 和 affix 文件的质量。如果选择使用 Hunspell 词干提取器，为了更好地处理特定领域的词，创建一个自定义词典是很必要的。虽然很多 Solr 用户因为 Hunspell 词干提取器的这些局限性而选择不使用它，但如果语种不是被 Solr 本地支持的，或者需要为应用建立特定领域的词典，那么 Hunspell 词干提取器很值得一试。

截至目前，本章已经介绍了 Solr 中可用的很多种特定语种文本分析组件，下面介绍如何组合地使用它们来搜索多语种内容。

## 14.6 在多语种中搜索内容

很多企业需要跨语种搜索内容。目前为止，我们已经看到了 Solr 跨语种的内容处理能力，但是这仅限于每次在单一的语种中配置搜索。虽然它可以创建一个英语字段或者德语字段，但如果需要同时跨这两种语言进行搜索呢？

一个文档集可能有不同的语种，甚至在一个文档或字段中就可能有多种语种。本节将介绍一些多语种内容搜索的技术，有三种主要的方法来实现这些类型的多语种搜索功能。

- 为每种语言创建一个独立的字段，跨每个字段查询。
- 在相同字段名上建立并配置多个索引，为每个索引配置字段来处理不同的语种。
- 设置一种能够同时支持跨语种索引和搜索的字段类型。

每种技术的复杂程度不同，适用的情况也不同，在接下来的小节中将详细介绍这些技术。

### 14.6.1 每种语言一个独立字段

跨多种特定语种字段的搜索在 Solr 的多语种配置中是最简单和最常见的。在一个较高的层次上，它需要预先知道哪个语种或者哪几个语种和搜索内容相关，并确认内容在索引时已经被导入到这些字段中。在搜索时，需要明确应用程序要跨哪些语种搜索，必须将它们对应的字段包含在查询语句中。图 14.4 为 field-per-language

多语种搜索的配置提供了索引和查询模型。

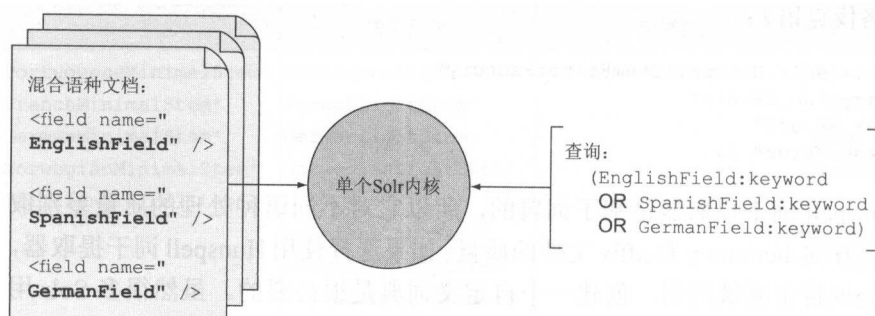


图 14.4 使用 field-per-language 模型配置多语种搜索的索引和查询

从图 14.4 中可以看出，field-per-language 多语种搜索模型不仅要求按照语种将内容分为各个字段，还要求在这些字段上复制查询关键词。这种为每个语种维护独立字段的方法是前面提到的三种设定方法中最简单的一个，但是当添加更多特定语种的字段时，它的扩展性是最差的。扩展性问题体现在以下两个方面：

- 如果在查询时不能将查询语句减少到少数特定语种，就很可能需要在文档中定义的所有特定语种字段中展开搜索，而对每一个额外字段的搜索都会降低查询速度。
- 如果在一个单独的文档中需要多语种支持，最后可能不得不将该文档的内容复制到多个字段中，这增加了索引的规模。这样，应用程序的查询速度会降低，并且如果不增加额外的索引分片，应用程序可以搜索的内容量也会降低。

## Field-per-language 介绍

### 优点

- 实现简单，可以开箱即用。
- 内置支持利用 DisMax-style 查询解析器实现跨字段搜索。
- 开箱即用的语种检测可以将特定语种映射到对应字段。
- 可在一个单独的 Solr 内核中或者分片的 Solr 中配置使用。

### 缺点

- 多语种文档可能要求每个语种字段有双倍存储内容，这增加了索引规模，从而降低了搜索速度。
- 查询的语种数量增加时，搜索速度会减慢。
- 必须使用 DisMax-style 查询解析器或手动构建一个多字段查询。
- 不支持混合语种的邻近搜索和词组搜索。

因为大多数多语种搜索配置简单且复杂性有限，所以这种方法是 Solr 中最常用

的方法，也是处理多语种检索最有效的方法。

代码清单 14.2 展示了如何设置应用程序的模式来实现跨英语、西班牙语和法语三种语言的搜索。

代码清单 14.2 支持三个独立语种字段的模式

```
<fieldType name="text_english" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory"
      ignoreCase="true"
      words="lang/stopwords_en.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory"
      protected="protowords.txt"/>
    <filter class="solr.KStemFilterFactory"/>
  </analyzer>
</fieldType>

<fieldType name="text_spanish" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/
      stopwords_es.txt" format="snowball"/>
    <filter class="solr.SpanishLightStemFilterFactory"/>
  </analyzer>
</fieldType>

<fieldType name="text_french" class="solr.TextField"
  positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.ElisionFilterFactory" ignoreCase="true"
      articles="lang/contractions_fr.txt"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/
      stopwords_fr.txt" format="snowball"/>
    <filter class="solr.FrenchLightStemFilterFactory"/>
  </analyzer>
</fieldType>
...
<field name="id" type="string" indexed="true" stored="true" />
<field name="title" type="string" indexed="true" stored="true" />
<field name="content_english" type="text_english" indexed="true"
  stored="true"/>
<field name="content_spanish" type="text_spanish" indexed="true"
  stored="true" />
```



```
<field name="content_french" type="text_french" indexed="true"
  stored="true" />
...
```

在 schema.xml 文件中为每种语言设置一个字段后, 下一步就是为文档构建索引, 每个文档的内容都会被映射到合适的特定语种字段上。以下命令可以将文档加载到 Solr 中:

```
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/field-per-language/update
  -jar post.jar ch14/documents/field-per-language.xml
```

在开始搜索加载到 Solr 中的文档之前, 应该先了解一下这些文档加载到 Solr 后的结构, 代码清单 14.3 很好地介绍了一些包含著名书籍引文的文档。

#### 代码清单 14.3 文档集中每个语种一个单独字段

```
<doc>
  <field name="id">1</field>
  <field name="title">The Adventures of Huckleberry Finn</field>
  <field name="content_english">YOU don't know about me without you have read
    a book by the name of The Adventures of Tom Sawyer; but that ain't no
    matter. That book was made by Mr. Mark Twain, and he told the truth,
    mainly. There was things which he stretched, but mainly he told the truth.
  </field>
</doc>
<doc>
  <field name="id">2</field>
  <field name="title">Les Misérables</field>
  <field name="content_french">Nul n'aurait pu le dire; tout ce qu'on savait,
    c'est que, lorsqu'il revint d'Italie, il était prêtre.
  </field>
</doc>
<doc>
  <field name="id">3</field>
  <field name="title">Don Quixote</field>
  <field name="content_spanish">Demasiada cordura puede ser la peor de las
    locuras, ver la vida como es y no como debería de ser.
  </field>
</doc>
<doc>
  <field name="id">4</field>
  <field name="title">Proverbs</field>
  <field name="content_spanish">No la abandones y ella velará sobre
    ti, ámala y ella te protegerá. Lo principal es la sabiduría; adquiere
    sabiduría, y con todo lo que obtengas adquiere inteligencia.
  </field>
  <field name="content_english">Do not forsake wisdom, and she will
    protect you; love her, and she will watch over you. Wisdom is supreme;
    therefore get wisdom. Though it cost all you have, get understanding.
  </field>
```



```
<field name="content_french">N'abandonne pas la sagesse, et elle te
gardera, aime-la, et elle te protégera. Voici le début de la sagesse:
acquiers la sagesse, procure-toi le discernement au prix de tout ce que tu
possèdes.
</field>
</doc>
```

该代码清单中有两点重要内容需要注意。每个语种都有一个独立的内容字段 (content\_english、content\_french 和 content\_spanish)，虽然大多数文档只用了这些字段中的一个，但这种情况没有被限制。例如，在 id 为 4 的文档中，可以看到每个语种都有对应的翻译，因此可以用任何一种语言搜索它。这里并不要求三个字段中的文本翻译很精确；如果一个文档的不同部分可以跨语种进行划分，那么将内容中独立的部分对应到恰当的语种字段中是合乎逻辑的。

在 Solr 中构建好这些文档的索引后，可以进行最后一步：进行跨语种搜索。回顾第 7 章的内容，eDisMax 查询解析器让跨多字段搜索变得简单。代码清单 14.4 展示了如何轻松地在本节示例定义的语种字段上执行检索。

#### 代码清单 14.4 跨多个特定语种字段搜索

##### 查询请求

```
http://localhost:8983/solr/field-per-language/select?
fl=title&
defType=edismax&
qf=content_english content_french content_spanish&
q="he told the truth" OR "il était prêtre" OR "ver la vida como es"
```

##### 搜索结果

```
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "response":{"numFound":3,"start":0,"docs":[
    {
      "title":["The Adventures of Huckleberry Finn"]},
    {
      "title":["Don Quixote"]},
    {
      "title":["Les Misérables"]}]
  }}
```

如果执行这个查询，Solr 在搜索结果中将返回三本书对应的文档：Don Quixote（和 content\_spanish 字段中的短语 "ver la vida como es" 匹配）、The Adventures of Huckleberry Finn（和 content\_english 字段中的短语 "he told the truth" 匹配），以及 Les Misérables（和 content\_french 字段中的短语 "ver la vida como es" 匹配）。这个示例表明进行跨语种检索是相对简单的，其

中每个文档包含一种语言的独立的特定语种字段。

但是如果单个文档需要包含多语种会怎样呢？代码清单 14.5 展示了在一个多语种文档中执行搜索的示例。

#### 代码清单 14.5 搜索每个字段一种语言的多语种文档

##### 查询请求 1

```
http://localhost:8983/solr/field-per-language/select?
  fl=title&
  defType=edismax&
  qf=content_english content_french content_spanish&
  q="wisdom"
```

##### 查询请求 2

```
http://localhost:8983/solr/field-per-language/select?
...
q="sabiduría"
```

##### 查询请求 3

```
http://localhost:8983/solr/field-per-language/select?
...
q="sagesse"
```

搜索结果(查询请求 1~3 的结果相同)

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 1},
  "response": { "numFound": 1, "start": 0, "docs": [
    {
      "title": ["Proverbs"]}
  ]
}}
```

该代码清单中返回的文档包含内容的三种翻译：分别是 `content_english` 字段、`content_spanish` 字段和 `content_french` 字段（请参考代码清单 14.3 中关于这些字段的全文）。因此，这三个查询（以 `wisdom` 在三种语言中的任何一种翻译的查询为例）都会搜索到同一份文档。以上表明了如何通过多语种翻译来支持文档搜索，尽管没有确切要求每个字段中的内容都包含精确的翻译。假如文档中不同部分的语种不同，如果能事先判断出每个部分的语种，就能将这些部分划分到特定语种的字段中，这将是很有意义的。

正如本节所讲，很容易通过将多语种内容划分到特定语种字段中并进行跨字段搜索，从而支持多语言内容。直到有多语种支持时，每个语种模型的独立字段才会发挥作用。由于每个查询会跨多种特定语种字段进行搜索，如果需要搜索的语种或字段数量变得很多，就会显著影响查询速度。如果出现这种情况，可能就要考虑接下来的章节所讲的更先进的多语种搜索方法。

## 14.6.2 每个语种构建单独的索引

除了在搜索索引中创建多个特定语种字段并在查询时跨这些字段进行搜索，基于单个字段的搜索也可以达到同样的目的。常见方法是每个语种创建单独的 Solr 索引（Solr 内核）。相对于在相同的索引中为每个语种创建单独的 Solr 字段（上一节中展示的），为每个语种创建独立的 Solr 内核可以通过改变引用自每个索引中字段的字段类型来处理不同语种，同时每个索引中保存相同的字段名称。这使查询得以保持简单。相对于在同一个 Solr 索引中进行多字段扩展查询，通过跨多 Solr 内核（CPU 内核或服务器）实现不同语种的并行搜索，可以使得每次只需搜索一个单独的字段，提高了查询的速度。图 14.5 展示了这种多语种搜索实现的架构。

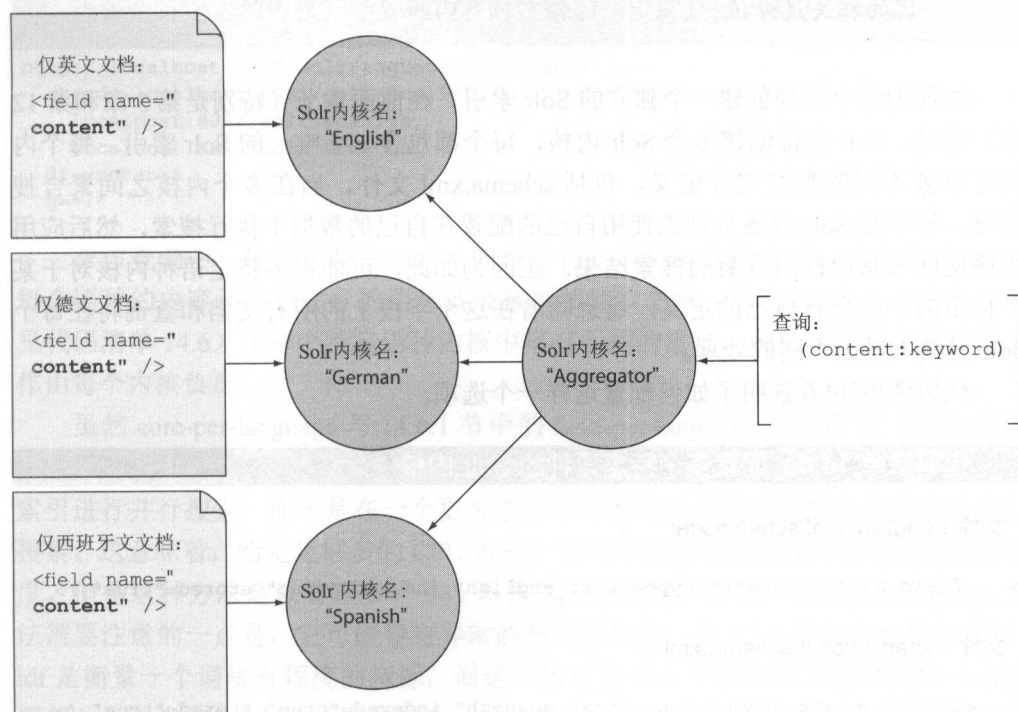


图 14.5 使用 core-per-language 模型实现多语种搜索的索引和查询配置图

从图 14.5 可以看出，与上一节中讨论的 field-per-language 方法相比，core-per-language 实现较为复杂。它需要管理单独的 Solr 索引，并基于内容的语种手动地向它们引导内容。查询是相当简单的，因为可以通过配置 Solr 将多个 Solr 索引的搜索结果自动汇总，但整体的架构肯定需要更多动态的管理。

## Core-per-language 介绍

## 优点

- 快速跨语种搜索——并行搜索每个语种的内容。
- 支持任何查询解析器，因为它只在单个字段上搜索。

## 缺点

- 管理多个 Solr 内核增加了系统复杂性。
- 在加载到 Solr 之前，必须根据语种手动将文档分片。
- 必须将多语种文档复制到多个内核中，才能支持多语种文档。
- 每个语种都要计算相关度，然后和其他语种的相关度合并，这会产生次优的相关度分值。

如何为每个语种创建一个独立的 Solr 索引？在前面章节（特别是第 3 章和第 12 章）讲过，Solr 支持创建多个 Solr 内核，每个都包含一个唯一的 Solr 索引。每个内核可以被不同的配置文件定义，包括 schema.xml 文件。当在多个内核之间聚合搜索时，每个是 Solr 内核分别先使用自己的配置在自己的数据中执行搜索，然后应用程序返回所有内核组合后的搜索结果。正因为如此，可能很多特定语种内核对于某个特定的字段会有自己的定义，这意味着在这个字段上的所有文档和查询将在每个 Solr 内核上做出不同的处理。

代码清单 14.6 说明了如何配置这样一个选项。

代码清单 14.6 配置多个特定语种的 Solr 内核

文件：english/conf/schema.xml

...

<field name="content" type="text\_english" indexed="true" stored="true" />

...

文件：spanish/conf/schema.xml

...

<field name="content" type="text\_spanish" indexed="true" stored="true" />

...

文件：french/conf/schema.xml

...

<field name="content" type="text\_french" indexed="true" stored="true" />

内核定义：(core.properties 文件)

\$SOLR\_INSTALL/example/solr/english/core.properties

\$SOLR\_INSTALL/example/solr/spanish/core.properties

\$SOLR\_INSTALL/example/solr/french/core.properties

\$SOLR\_INSTALL/example/solr/aggregator/core.properties

每个语种内核使用一个语种专用 schema。

聚合器用于搜索其他内核（包含没有数据的在内）。

相同的  
字段名  
用于定  
义不同  
的语种。

在这个特殊的示例中，每个语种都定义了独立的 Solr 内核，每个内核使用自定义的模式，这种模式包含一个 content 字段，用于使用特定语种的字段类型。在运行本节示例之前，需要像代码清单 14.6 一样配置特定语种的 Solr 内核，配置好后就可以向 Solr 内核中加载内容：

```
cd $SOLR_IN_ACTION/example-docs/  
java -jar -Durl=http://localhost:8983/solr/english/update post.jar  
    ➡ ch14/documents/english.xml  
java -jar -Durl=http://localhost:8983/solr/spanish/update post.jar  
    ➡ ch14/documents/spanish.xml  
java -jar -Durl=http://localhost:8983/solr/french/update post.jar  
    ➡ ch14/documents/french.xml
```

当每个特定语种内核开始处理内容时，最后一步就是输入查询语句：

```
http://localhost:8983/solr/aggregator/select?  
shards=localhost:8983/solr/english,  
    localhost:8983/solr/spanish,  
    localhost:8983/solr/french&  
df=content&  
q=*:*
```

在该查询中，使用参数 shards 向每个特定语种的内核发送一个分布式请求，每个语种的内容和其他并行的语种就会被单独搜索出来。因为 content 字段（参见代码清单 14.6）在每个特定语种内核中都有不同的定义，所以处理每个语种的工作由每个内核负责。

虽然 core-per-language 与 14.6.1 节中的 field-per-language 方法类似（都允许设置跨所有语种或部分语种的查询），但它具有额外的优点，即它可以跨多个更小的索引进行并行搜索，而不是在一个更大的索引中跨数目不断增长的多个字段来进行搜索。这意味着，给定足够多的 CPU 内核，这种方法通常会根据其规模提高查询速度。但是这种方法是管理多个 Solr 内核的复杂性为代价的。core-per-language 方法需要注意的一点是，它可能导致异常的相关度得分。从 3.2.3 节的内容可以知道，idf 是衡量一个词稀有程度的指标，而这个指标是 Solr 中默认相关度算法的一个重要部分。因为 idf 是基于所有文档在一个单独的 Solr 索引（Solr 内核）中的统计值，所以如果词在每个 Solr 内核中随机分布，那么它的效果是最好的。在根据语种划分内容的情况下，显然不是随机的，内容的分片方式可能会在不经意间导致一种语言的相关度得分高于另一种语言。

此外，field-per-language 同一文档中处理多语种的能力很难复制到 core-per-language 中。因为在本节的 core-per-language 的示例中，每一份文档只包含一个 content 字段，所以每份文档只能处理每个 Solr 内核中的一种语言配置。解决这



一限制的唯一途径是向不同的 Solr 内核中加载相同的文档，但并不推荐这样做，因为它会让文档在任何搜索的元数据（facets、hits 等）中计算两次。处理每个文档中的多语种问题在很多系统中不是必需的，但是在实现这个方法之前上述问题是值得考虑的。

除了 field-per-language 和 core-per-language 这两个方法，也可以通过允许单个字段支持多语种来实现多语种搜索，这个方法将会在下一节详述。

### 14.6.3 支持多语种的单个字段

为每个语种都管理一个独立的 Solr 内核会增加应用程序管理的复杂度，因为它需要创建的一个自定义的分片方法，为每个语种将文档分配到独立的 Solr 内核中。此外，如果文档数量较少，使用分布式搜索的聚合开销会降低查询速度。从 14.6.2 节可以看出，core-per-language 方法对包含多语种的单个文档的支持也存在难度。每个语种有一个独立字段会导致查询变得更复杂，如果使用的查询解析器不在 DisMax 查询解析器的系列（比如 eDisMax 查询解析器）中，就必须手动地在每个特定语种字段中输入查询。

在某些情况下这将变得更简单，但需要将这两种方法的主要优点结合起来：一个跨多语种的单个字段，不需要管理特定语种的 Solr 内核和支持多语种文档的能力。这可以通过创建一个支持语种配置的任何组合的多值字段类型来实现。

#### 支持多语种的单个字段的介绍

##### 优点

- 在单个字段上支持任意语种间的组合。
- 可以在一个 Solr 内核或分片 Solr 中设置运行。
- 支持任何查询解析器，因为它只在单个字段上搜索。
- 支持真正的多语种搜索，包括混合语种短语。
- 不需要对每个语种的内容进行双存储，减少了索引规模并提高了搜索速度。
- 允许在查询时动态组合不同语种。
- 仍然支持语种识别（可以通过本章的示例代码实现）。

##### 缺点

- 不是开箱即用的，需要自定义代码（包含在本章内）。
- 每个文档字段和查询词都必须带有语种中能够被分析或增加复杂性的地方。
- 如果索引和查询时选择的语种不匹配，将存在词无法发现的潜在问题。

每个字段支持多语种并不是 Solr 提供的开箱即用的功能（目前来讲），但是可以通过一些方法来实现这个功能。实现这个功能的方法有很多。最简单的一种方法是在 `schema.xml` 文件中定义的相同 `TextField` 分析器链中放置多个词干提取器（词干过滤器）。这种方法的缺点是之前的词干提取器在遇到后面的词干提取器之前可能会破坏词素的原始格式，导致在索引中创建不想要的或者错误的词素。

可以通过创建一个自定义的 `TokenFilter` 来解决这个问题，它可以在将原始数据输入单独的词干提取器之前，缓冲词素并创建多个原始词素的副本。通过这个方法，最终会得到多种词素流，然后不得不堆叠或连接它们。如果只需要支持一些预定义的语种，这个方法就很有效，这些语种只需要在分析链中的一个地方置换出 `TokenFilter`。

如果想通过不同的配置（`TokenFilters` 及 `CharFilters`、`Tokenizers` 的顺序都是变量）来支持很多不同的语种，就需要一个更灵活的解决方案。为了提供解决这种问题的一个示例，本章创建了一个特殊的字段类型——`MultiTextField`——它将有效地支持把不同的字段类型堆叠在一个字段中。在下一节中，将展示这个字段类型如何起作用，以及怎样利用它支持相同字段中的高度灵活的多语种搜索。

#### 14.6.4 创建一个字段类型来处理支持多语种的单个字段

当向 Solr 加载文档或查询时，每个字段的内容会根据这个字段定义的字段类型被分析。这个字段类型可以包含每个模型中的一个分析器：索引模型、查询模型和 `multiTerm` 模型（用来支持通配符查询）。在多语种内容的情况下，如果可以根据内容指定多个分析器（每个语种一个），它将会很有用，因为有时候会需要每个语种都有一个完全独立的分析器链（通过配置独立的 `CharFilter`、`Tokenizer` 和 `TokenFilter`）。

在这种情况下，需要指定一种在索引每个字段和查询每个词时的特定语种。图 14.6 提供了一个在这种配置下的文档和查询的模型。

从图 14.6 可以看出，在索引和查询时，内容的语种必须被指定。虽然很多语种都可以根据意愿组合起来，并且它们在各个文档和查询中可以是不同的，但是要实现这一点就需要告诉特殊的字段类型用哪个语种设置去分析文档。对于本章中的操作，语种代码清单就附在内容前面。



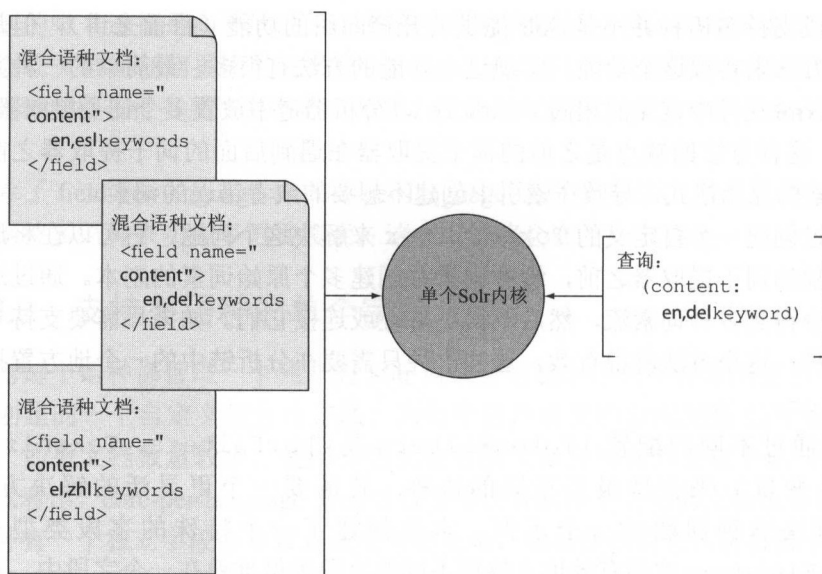


图 14.6 在同一个 Solr 内核中将多语种组合到一个字段来实现多语种搜索的索引和查询配置图

一个示例文档如下所示：

```
<doc>
  <field name="content">
    en,de,fr|this field contains English, dieses feld enthält Deutsch,
    et ce champ contient du français
  </field>
</doc>
```

同样，一个查询示例如下所示：

```
/select?q=en|contains OR de|enthält OR en,fr|contient
```

在这个索引示例中，代码清单中内容前面的语种列表暗示了内容应被处理为英语(en)、德语(de)和法语(fr)。同样，在查询示例中，第一个词会被处理为一个英语词，第二个词会被处理为一个德语词，第三个词会同时运行英语和法语分析器。为了完成这个功能，需要创建一个可以根据需要在不同分析器间进行转换和组合的分析器。代码清单 14.7 展示了通过自定义 TextField 类和 Analyzer 类来创建这种机制的第一步。

#### 代码清单 14.7 实现 MultiTextField 类和 Analyzer 类

```
public class MultiTextField extends TextField {
  ... //Skipped code which reads in settings, available in full source
```

```

@Override
protected void init(IndexSchema schema, Map<String,String> args) {
    super.init(schema, args);

    MultiTextFieldSettings indexSettings = new MultiTextFieldSettings();
    indexSettings.analyzerMode = AnalyzerModes.index;
    MultiTextFieldSettings querySettings = new MultiTextFieldSettings();
    querySettings.analyzerMode = AnalyzerModes.query;
    MultiTextFieldSettings multiTermSettings = new MultiTextFieldSettings();
    multiTermSettings.analyzerMode = AnalyzerModes.multiTerm;

    ...

    MultiTextFieldAnalyzer indexAnalyzer = new
    MultiTextFieldAnalyzer(schema, indexSettings);
    MultiTextFieldAnalyzer queryAnalyzer = new
    MultiTextFieldAnalyzer(schema, querySettings);
    MultiTextFieldAnalyzer multiTermAnalyzer = new
    MultiTextFieldAnalyzer(schema, multiTermSettings);

    this.setAnalyzer(indexAnalyzer);
    this.setQueryAnalyzer(queryAnalyzer);
    this.setMultiTermAnalyzer(multiTermAnalyzer);
}

}

public class MultiTextFieldAnalyzer extends Analyzer {

    protected IndexSchema indexSchema;
    protected MultiTextFieldSettings settings;

    public MultiTextFieldAnalyzer(IndexSchema indexSchema,
    MultiTextFieldSettings settings) {
        super(new PerFieldReuseStrategy());
        this.settings = settings;
        this.indexSchema = indexSchema;
    }

    @Override
    public TokenStreamComponents createComponents(String fieldName,
    Reader reader) {

        MultiTextFieldTokenizer multiTokenizer = new MultiTextFieldTokenizer(
            indexSchema, reader, fieldName, Settings);

        Tokenizer source = multiTokenizer;
        TokenStream result = multiTokenizer;
        if (Settings.removeDuplicates){
            result = new RemoveDuplicatesTokenFilter(multiTokenizer);
        }

        return new TokenStreamComponents(source, result);
    }

}

```

从这个代码清单中可以看出, MultiTextField 类和 MultiTextFieldAnalyzer 类没有做什么令人印象深刻的事情。因为 TextField 包含了三个独

立的分析器——一个用于索引内容，一个用于查询，一个用于进行通配符和类似查询——所以 `MultiTextField` 类确保了 `MultiTextFieldAnalyzer` 类的一个独立副本在这些配置中被指定。由于 `MultiTextFieldAnalyzer` 类需要为每个子字段载入合适的分析器，因此在子字段中分配知道选择哪些分析器（`index`、`query` 或 `multiTerm`）的不同的 `MultiTextFieldAnalyzer` 是必要的。另一个需要注意的重要问题是 `MultiTextField` 和 `MultiTextFieldAnalyzer` 将 `IndexSchema` 传递给 `MultiTextFieldTokenizer`，并向它提供访问所有其他字段类型和配置的接口。通常情况下，分词器不能访问 `IndexSchema` 或其字段类型，但是因为 `MultiTextField` 是一种元字段，所以它的分析器和分词器需要访问分析管道可能被请求的其他可用字段的配置。

在之前的代码清单中，已被初始化的 `MultiTextFieldTokenizer` 类是最难的部分。代码清单 14.8 介绍了 `MultiTextFieldTokenizer` 类。

代码清单 14.8 实现 `MultiTextField` 的分词器

```
public class MultiTextFieldTokenizer extends Tokenizer {
    protected String fieldName;
    protected IndexSchema indexSchema;
    protected MultiTextFieldSettings settings;
    protected LinkedHashMap<String, Analyzer> namedAnalyzers;
    protected MultiTextFieldInput multiTextInput;

    private CharTermAttribute charTermAttribute;
    private OffsetAttribute offsetAttribute;
    private TypeAttribute typeAttribute;
    private PositionIncrementAttribute positionAttribute;
    private LinkedList<Token> tokens;
    private Integer startingOffset;

    protected MultiTextFieldTokenizer(IndexSchema indexSchema, Reader input,
        String fieldName, MultiTextFieldSettings settings) {
        super(input);
        this.indexSchema = indexSchema;
        this.fieldName = fieldName;
        this.settings = settings;
        init();
    }

    private void init() {
        charTermAttribute = addAttribute(CharTermAttribute.class);
        offsetAttribute = addAttribute(OffsetAttribute.class);
        typeAttribute = addAttribute(TypeAttribute.class);
        positionAttribute = addAttribute(PositionIncrementAttribute.class);
    }

    @Override
```

构造器读入原始文本。

MultiTextFieldInput 从文本中分隔出请求的字段类型。

3

在 reset 方法中对输入进行预处理。

2

允许在指定的搜索词项上使用不同的分析器。

4

```

public void reset() throws IOException {
    super.reset();
    this.tokens = null;
    if (this.multiTextFieldInput == null) {
        this.multiTextFieldInput = new MultiTextFieldInput(
            this.input, this.settings.keyFromTextDelimiter,
            this.settings.multiKeyDelimiter);
    } else {
        this.multiTextFieldInput.setReader(this.input);
    }
    this.namedAnalyzers = getNamedAnalyzers();
    this.startingOffset = this.multiTextFieldInput.StrippedIncomingPrefixLength
        >= 0 ? this.multiTextFieldInput.StrippedIncomingPrefixLength : 0;
}

private LinkedHashMap<String, Analyzer> getNamedAnalyzers() {
    LinkedHashMap<String, Analyzer> namedAnalyzers =
        new LinkedHashMap<String, Analyzer>();

    FieldType fieldType;
    for (int i = 0; i < this.multiTextFieldInput.Keys.size(); i++) {

        String fieldTypeName = this.multiTextFieldInput.Keys.get(i);
        if (this.settings.fieldMappings != null) {
            fieldTypeName = this.settings.fieldMappings
                .get(this.multiTextFieldInput.Keys.get(i));
        }

        fieldType = this.indexSchema.getFieldTypeByName(fieldTypeName);
        if (fieldType != null) {
            if (this.settings.analyzerMode == AnalyzerModes.query) {
                namedAnalyzers.put(fieldTypeName, fieldType.getQueryAnalyzer());
            } else if (this.settings.analyzerMode == AnalyzerModes.multiTerm) {
                namedAnalyzers.put(fieldTypeName,
                    ((TextField) fieldType).getMultiTermAnalyzer());
            } else {
                namedAnalyzers.put(fieldTypeName, fieldType.getAnalyzer());
            }
        } else {
            if (!this.settings.ignoreMissingMappings) {
                throw new SolrException(SolrException.ErrorCode.BAD_REQUEST,
                    "Invalid FieldMapping requested: '"
                        + this.multiTextFieldInput.Keys.get(i) + "'");
            }
        }
    }
}

if (namedAnalyzers.size() < 1) {
    if (this.settings.defaultFieldType != null
        && this.settings.defaultFieldType.length() > 0) {
        if (this.settings.analyzerMode == AnalyzerModes.query) {
            namedAnalyzers.put(

```

如果未指定分析器, 则使用默认字段类型。

```

        "",
        this.indexSchema.getFieldTypeByName(
            this.settings.defaultFieldName).getQueryAnalyzer());
    } else if (this.settings.analyzerMode == AnalyzerModes.multiTerm) {
        namedAnalyzers.put("", ((TextField) this.indexSchema
            .getFieldTypeByName(this.settings.defaultFieldName))
            .getMultiTermAnalyzer());
    } else {
        namedAnalyzers.put(
            "",
            this.indexSchema.getFieldTypeByName(
                this.settings.defaultFieldName).getAnalyzer());
    }
}

if (namedAnalyzers.size() == 0) {
    throw new SolrException(SolrException.ErrorCode.BAD_REQUEST,
        "No FieldMapping was Requested, and no DefaultField"
        + " is defined for MultiTextField '" + this.fieldName
        + "'. A MultiTextField must have one or more "
        + "FieldTypes requested to execute a query.");
}

return namedAnalyzers;
}

```

```

@Override
public boolean incrementToken() throws IOException {
    if (this.tokens == null) {
        String data = convertReaderToString(this.multiTextInput.Reader);
        if (data.equals("")) {
            return false;
        }

        this.tokens = mergeToSingleTokenStream(
            createPositionsToTokensMap(this.namedAnalyzers, data));

        if (this.tokens == null) {
            return false;
        }
    }

    if (tokens.isEmpty()) {
        this.tokens = null;
        return false;
    } else {
        clearAttributes();
        Token token = tokens.removeFirst();

        this.charTermAttribute.copyBuffer(token.buffer(), 0, token.length());
        this.offsetAttribute.setOffset(token.startOffset(), token.endOffset()
            + this.startingOffset);
        this.typeAttribute.setType(token.type());
        this.positionAttribute.setPositionIncrement(

```

5  
所有词元在  
increment-  
Token() 第  
一次调用时  
进行缓存。

```

        token.getPositionIncrement());
    return true;
}

private SortedMap<Integer, LinkedList<Token>>
    createPositionsToTokensMap (LinkedHashMap<String, Analyzer>
        namedAnalyzers, String text) throws IOException {

    SortedMap<Integer, LinkedList<Token>> tokenHash =
        new TreeMap<Integer, LinkedList<Token>>();
    for (Map.Entry<String, Analyzer> namedAnalyzer :
        this.namedAnalyzers.entrySet()) {

        String subFieldName = (this.fieldName + " "
            + namedAnalyzer.getKey()).trim();

        addTokenStreamForFieldType(tokenHash, namedAnalyzer.getValue()
            .tokenStream(subFieldName, new StringReader(text)));
    }

    return tokenHash;
}

private void addTokenStreamForFieldType(
    SortedMap<Integer, LinkedList<Token>> tokenHash,
    TokenStream tokenStream) throws IOException {

    tokenStream.reset();
    int position = 0;

    CharTermAttribute charTermAtt = null;
    PositionIncrementAttribute posIncrAtt = null;
    OffsetAttribute offsetAtt = null;
    TypeAttribute typeAtt = null;

    if (tokenStream.hasAttribute(CharTermAttribute.class)) {
        charTermAtt = tokenStream.getAttribute(CharTermAttribute.class);
    }

    if (tokenStream.hasAttribute(PositionIncrementAttribute.class)) {
        posIncrAtt =
            tokenStream.getAttribute(PositionIncrementAttribute.class);
    }

    if (tokenStream.hasAttribute(OffsetAttribute.class)) {
        offsetAtt = tokenStream.getAttribute(OffsetAttribute.class);
    }

    if (tokenStream.hasAttribute(TypeAttribute.class)) {
        typeAtt = tokenStream.getAttribute(TypeAttribute.class);
    }

    for (boolean hasMoreTokens = tokenStream.incrementToken();
        hasMoreTokens; hasMoreTokens = tokenStream.incrementToken()) {

        String multiTermSafeType = null;

```

生成一张词元的位置地图，从每个 TokenStream 中合并词元。

每个字段的 TokenStream 组件都会缓存，因此必须模拟子字段。

6  
从每个请求的分析器中取得一个单独的 TokenStream。

```

if (charTermAtt == null
    || offsetAtt == null
    || (typeAtt == null &&
        this.settings.analyzerMode != AnalyzerModes.multiTerm)) {
    return;
}

if (typeAtt != null) {
    multiTermSafeType = typeAtt.type();
}

Token clone = new Token(charTermAtt.toString().trim(),
    offsetAtt.startOffset(), offsetAtt.endOffset(), multiTermSafeType);
position += ((posIncrAtt != null) ?
    posIncrAtt.getPositionIncrement() : 1);

if (!tokenHash.containsKey(position)) {
    tokenHash.put(position, new LinkedList<Token>());
}

tokenHash.get(position).add(clone);
}
tokenStream.close();
}

private static LinkedList<Token> mergeToSingleTokenStream
    (SortedMap<Integer, LinkedList<Token>> tokenHash) {

    LinkedList<Token> result = new LinkedList<Token>();

    int currentPosition = 0;
    for (int newPosition : tokenHash.keySet()) {
        int incrementTokenIndex = result.size();

        LinkedList<Token> brothers = tokenHash.get(newPosition);
        int positionIncrement = newPosition - currentPosition;

        for (Token token : brothers) {
            token.setPositionIncrement(0);
            result.add(token);
        }

        if (result.size() > incrementTokenIndex
            && result.get(incrementTokenIndex) != null) {
            result.get(incrementTokenIndex)
                .setPositionIncrement(positionIncrement);
        }

        currentPosition = newPosition;
    }

    return result;
}
}

```

7  
将词元地图  
上的词元位  
置合并成最  
终的一个  
TokenStream。



MultiTextFieldTokenizer 类的代码量之大让它看起来很复杂，但是它的实现可以被分解为以下几个简单的步骤。

1. 像所有的分词器 (Tokenizers) 一样，一个字段名和一个读取器 ❶ 可通过构造函数或基类中的 `setReader()` 方法来传入。`setReader` 方法允许对象被不同的读取器复用，而不需要为每个请求创建一个新的 `Tokenizer` 对象。
2. 像所有的分词器一样，在每次读取器改变时，`reset()` 方法 ❷ 都会被调用。读取器包含了要被分析的文档，所以它随着新文档的传入而改变。`MultiTextTokenizer` 使用 `reset()` 方法解析读取器中的文档，使用 ❸ `MultiTextFieldInput` 类 (未显示出来) 来分隔字段内容的语种映射。然后，在 `getNamedAnalyzers()` 方法 ❹ 中创建了一个分词器列表，该列表在随后的处理中会被用到，它与读取器中开始内容中的字段相对应。
3. 当 `incrementToken()` ❺ 在 `MultiTextFieldTokenizer` 类中被调用时，对于每个所要求的字段类型，都会将要传入读取器中的文档的副本传给 `addTokenStreamForFieldType()` ❻。对于每个请求的字段类型，该方法可以为其获得一个分析器并调用 `tokenStream()`，最后利用分析器 `TokenStreamComponents` (`CharFilters`、`Tokenizer` 和 `TokenFilters`) 返回一个词素流 (`TokenStream`)。
4. 每一个被处理的 `TokenStreams` (每个请求的字段类型都有一个) 都会被堆叠/合并到 `mergeToSingleTokenStream()` ❼ 中的一个 `TokenStream` 中。这种方法根据每个词素的位置增量来决定词素最终被叠放到词素流中的位置。合并后的 `TokenStream` 被保存为一个实例变量，然后被用于在每次后续调用 `incrementToken` 方法后内部处理每个词素。

完成这些步骤后，`MultiTextFieldTokenizer` 可以根据一个或更多其他的文本字段来动态改变它的分析链。本章专门使用它的这项功能来请求一个或更多特定语种字段，但需要注意的是，这个功能并不局限于语种的变化；它还可以在正在运行的配置中的合适字段上改变或结合任何文本分析配置。这是一个强大而先进的功能，如果在索引和查询时没有请求相同的 (动态的) 分析，则会导致意外的搜索结果。

代码清单 14.9 展示了如何使用 `MultiTextField` 类，通过特定语种字段类型中定义的分析器在单个字段中进行多语种索引和搜索。

## 代码清单 14.9 在相同字段中实现多语种索引和查询

schema.xml

```
...
<field name="content" type="multiText" indexed="true" stored="true"/>
...
<fieldType name="multiText"
    class="sia.ch14.MultiTextField" sortMissingLast="true"
    defaultFieldType="text_general"
    fieldMappings="en:text_english,
                  es:text_spanish,
                  fr:text_french"/>
...
```

文档

```
<doc>
  <field name="id">1</field>
  <field name="title">Proverbs</field>
  <field name="content">
    en,fr,es|No la abandones y ella velará sobre ti, ámalala y ella te
    protegerá. Voici le début de la sagesse: acquiers la sagesse. Though
    it cost all you have, get understanding.
  </field>
</doc>
```

添加文档

```
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/multi-language-field/update
    -jar post.jar ch14/documents/multi-language-field.xml
```

查询请求

```
http://localhost:8983/solr/multi-language-field/select?
  fl=title&
  df=content&
  q=en,fr,es|abandon AND en,fr,es|understanding AND en,fr,es|sagess
```

搜索结果

```
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "response":{ "numFound":1, "start":0, "docs": [
    {
      "title": ["Proverbs"]}
  ]
}}
```

这个查询成功地匹配了 abandon、understand 和 sagess、尽管这个文档中包含的是 abandones（西班牙语）、understanding（英语）和 sagesse（法语）。因为在索引和查询时，每个被映射到特定语种的字段都会被请求（通过使用 en、fr、es 这些前缀），所以这些语种的任何变化都将被匹配——前提是为每个语种设定单独的字段或者单独的 Solr 内核。

如果想对 `MultiTextField` 进行更深入的实践，可以研究一下 Solr 的管理界面里的分析页面，它很好地展示了 Solr 内部的工作原理。图 14.7 演示了 Solr 如何分别使用英语文本字段和西班牙语文本字段来处理一个查询，以及 `MultiTextField` 如何将相同文档的输出进行融合。

**1 英文字段**

Field Value (Index)  
the schools, las escuelas

Analyse Fieldname / FieldType: text\_en ⓘ

ST	the	schools	las	escuelas
SF		schools	las	escuelas
LCF		schools	las	escuelas
EPF		schools	las	escuelas
KMF		schools	las	escuelas
PSF		school	la	escuela

**2 西班牙文字段**

Field Value (Index)  
the schools, las escuelas

Analyse Fieldname / FieldType: text\_es ⓘ

ST	the	schools	las	escuelas
LCF	the	schools	las	escuelas
SF	the	schools		escuelas
SLSF	the	schools		escuel

**3 英文与西班牙文混合的多文本字段**

Field Value (Index)  
en,es|the schools, las escuelas

Analyse Fieldname / FieldType: multiText ⓘ

RDTF	the	school	schools	la	escuela	escuel
------	-----	--------	---------	----	---------	--------

图 14.7 使用 Solr 的分析页面将一个英语分析器和西班牙语分析器的输出结果融合到 `MultiTextField` 字段的示例。第一个方框代表一个英语字段类型，第二个代表一个西班牙语字段类型，第三个是 `MultiTextField`，它使用英语和西班牙语的字段类型去分析文本

从图 14.7 可以看出，如果从 `MultiTextField` 映射到一个英语字段类型和一个西班牙语字段类型，那么最终输出结果是这两种字段类型合并后的词素流。在这个示例中，输入的文本是 `the schools, las escuelas`。英语字段类型的分析器将这个文本转变为三个词素：`school`、`la` 和 `escuela`。西班牙语的字段类型分析器将这个文本转变为三个不同词素：`the`、`schools` 和 `escuel`。明显地，这些特定语种分析器在处理其他语言时效果不好。当使用 `MultiTextField` 将多语种融合起来时，会发现每个语种的变化都呈现在了最后的 `TokenStream` 中：`the`、`school`、`schools`、`la`、`escuela`、`escuel`。这些词素不仅仅被附加在词素流中，

它们还被堆叠了起来，可以使用每个独立的分析器（此示例中是英语和西班牙语分析器）的位置偏移量来进行跨语种短语搜索。如果为先前的查询开启冗长模式，分析页面的输出结果将如图 14.8 所示。

Field Value (Index)							
en,es the schools, las escuelas							
Analyse Fieldname / FieldType:		multiTextContent		<input checked="" type="checkbox"/> Verbose Output			
RTDF	text	the	school	schools	la	escuela	escuel
	raw_bytes	[74 68 65]	[73 63 68 6f 6f 6c]	[73 63 68 6f 6f 6c 73]	[6c 61]	[65 73 63 75 65 6c 61]	[65 73 63 75 65 6c]
	start	0	4	4	13	17	17
	end	9	17	17	22	31	31
	type	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>	<ALPHANUM>
	position	1	2	2	3	4	4

图 14.8 与图 14.7 一样，但对页面输出结果做了更详细的分析。可以看到词“school”和“schools”的位置是一样的，“escuela”和“escuel”的位置也是一样的

从图 14.8 中可以看出，词素 school 和 schools 都被定位到位置 2，词素 escuela 和 escuel 都被定位到位置 4。下面看一个堆叠示例，以更好地理解词素流：

[Position 1]	[Position 2]	[Position 3]	[Position 4]
the	School schools	la	Escuela escuel

在相同的词素流中堆叠多语种，这在一个字段中融合多语种方面有独特的优势：虽然它可能不是一个特别常见的用例，如果有类似本例中的混合语种内容，可以执行短语搜索或邻近搜索，如 "the school la escuela" 或 "school escuelas" ~ 2，将能找到混合语种的匹配结果，而与 field-per-language 和 core-per-language 的多语种处理方式不同。

在索引和查询时换出语种是很容易的，但需要确保选择了正确的语种。因为用户查询往往包含很少的文本，所以除非事先了解用户，否则很难确定用户在本次查询时使用的是哪种语种。很多网站都能够利用用户的一些资料来确定搜索语种，比如通过用户的 Web 浏览器确定请求语种，通过用户的 IP 地址确定用户的地理位置，或者通过搜索语句确定查询中最可能使用的语种。如果应用程序支持的语种较少，可以在查询时将它们全部指定，但这会产生额外的处理每个语种的性能开销。如果所有指定的语种都失败了，可以要求用户在查询时选择所需的语种。

除了在查询时确定语种，更重要的是在索引时为文档选择正确的语种。这可以在内容被加载到 Solr 之前完成，参见代码清单 14.9，但 Solr 也有内置功能来检测文档中的语种。下一节将会演示如何使用它。

## 14.7 语种识别

支持多语种内容的一个内在挑战是确定出现在每个文档中的语种。在很多应用中可能已经提前确定了语种，因为要求用户识别内容中的语种，或者由特定语种的网站或频道来传递文档。但在大多数情况下需面对的是很多跨多语种的未标记文档。Solr 有内置的语种识别库，可以帮助解决这个问题。

在开始这个示例之前，需要指出的是，语种识别算法的精确度总是随着可供分析文档数量的增加而增加。设想一下，一个文档包含短语“Lucene/Solr”，在这个文档属于什么语种呢？如果全部的文档内容是“Estoy aprendiendo Lucene/Solr”，那答案应该是西班牙语，然而，如果内容是“I am learning Lucene/Solr”，那么答案应该是英语。这个短语不足以确定文档的语种，但是当增加一些词的时候，开发者（和语种识别算法）就会更好地辨别语种。

因为识别一种语言需要文本块，因此当全部的文档都呈现出来时，通常在索引时比在查询时更容易识别语种。我们并不推荐使用语种识别算法来确定用户查询语句中的语种类型，因为在很多搜索应用中用户的查询语句都很短，不能为识别语种类型提供足够有意义的内容。鉴于很多应用（特别是基于 Web 的应用）可以通过其他方式（如通过用户点击的那个国家的网站或者通过用户的属性或 IP 地址来确定用户的位置）来确定用户的语种类型，所以通常不同担心在查询时基于用户的查询语句来进行语种识别的不可靠性问题。

### 14.7.1 语种识别更新处理器

为了支持文档中的语种识别，Solr 提供了两个特有的更新处理器：TikaLanguageIdentifierUpdateProcessor 和 LangDetectLanguageIdentifierUpdateProcessor。第一个利用 Apache Tika 中的语种识别库；第二个使用 Java 的开源语种识别库，它的精确率很高（对很多语种而言都在 99% 以上）。代码清单 14.10 演示了如何安装语种识别器来识别文档中的语种。

代码清单 14.10 基本语种识别的更新处理器功能

**solrconfig.xml**

```
...
<updateRequestProcessorChain name="langid">
  <processor class="org.apache.solr.update.processor.
    LangDetectLanguageIdentifierUpdateProcessorFactory">
    <lst name="invariants">
      <str name="langid.fl">content,content_lang1,content_lang2,
        content_lang3</str>
      <str name="langid.langField">language</str>
```

```

        <str name="langid.langsField">languages</str>
        ...
    </lst>
</processor>
...
</updateRequestProcessorChain>
...

<requestHandler name="/update" class="solr.UpdateRequestHandler">
    <lst name="invariants">
        <str name="update.chain">langid</str>
    </lst>
</requestHandler>
...

```

#### schema.xml

```

...
<field name="language" type="string" indexed="true" stored="true" />
<field name="languages" type="string" indexed="true" stored="true"
    multiValued="true"/>
...

```

在这个代码清单中有三处重要内容。首先，一个名为 langid 的更新处理器已被定义，一个常量被添加到 /update 请求处理器中来确保 langid 更新处理器在每次更新时都在运行。在 /update 处理器中定义常量并不是必须的，但是如果不定义，将不能保证更新处理器在每个文档中都运行。例如，可以选择性地定义默认的 update.chain，它会可选地在任一给定查询中被覆写：

```

<requestHandler name="/update" class="solr.UpdateRequestHandler">
    <lst name="defaults">
        <str name="update.chain">langid</str>
    </lst>
</requestHandler>

```

否则，update.chain 会作为一个查询字符串变量在每次请求中传递。

<http://localhost:8983/solr/langid/update?update.chain=langid>

第二，有两个独立的语种字段参数被定义：langid.langField 和 langid.langsField。文档中主要被检测到的语种类型将被写入定义的 langid.langField 中，每个被检测到的语种——每个字段对应一个——将会被映射到 langid.langsField。因此，虽然 langid.langField 作为一个单值字段很好，但是除非 langid.langsField 的 multiValued 属性在 schema.xml 文件中设置为真，否则它不会起任何作用。

最后一点重要内容在之前提到过，即 Solr 中有两种不同的语种识别器。代码清单 14.10 演示了 Java 实现的语种检测库 (LangDetectLanguageIdentifierUpdateProcessorFactory) 的使用，可以很容易地使用来自相同包中的 TikaLangu



ageIdentifierUpdateProcessorFactory 类来替代 Tika 操作。本章后面的部分将介绍 Java 中的语种检测库的使用。

定义好语种识别更新处理器后，我们来向 Solr 中提交一些文档。这里将再次使用代码清单 14.3 中的文档，但是因为字段名被重新定义，所以不再使用特定语种的字段。被修改之后的文档如代码清单 14.11 所示。

#### 代码清单 14.11 语种识别文档

```
<doc>
  <field name="id">1</field>
  <field name="title">The Adventures of Huckleberry Finn</field>
  <field name="content">YOU don't know about me without you have read
    a book by the name of The Adventures of Tom Sawyer; but that ain't no
    atter. That book was made by Mr. Mark Twain, and he told the truth,
    mainly. There was things which he stretched, but mainly he told the
    truth.
  </field>
</doc>
<doc>
  <field name="id">2</field>
  <field name="title">Les Misérables</field>
  <field name="content">Nul n'aurait pu le dire; tout ce qu'on savait,
    c'est que, lorsqu'il revint d'Italie, il était prêtre.
  </field>
</doc>
<doc>
  <field name="id">3</field>
  <field name="title">Don Quixote</field>
  <field name="content">Demasiada cordura puede ser la peor de las locuras,
    ver la vida como es y no como debería de ser.
  </field>
</doc>
<doc>
  <field name="id">4</field>
  <field name="title">Proverbs</field>
  <field name="content_lang1"> No la abandones y ella velará sobre ti,
    ákala y ella te protegerá. Lo principal es la sabiduría; adquiere
    sabiduría, y con todo lo que obtengas adquiere inteligencia
  </field>
  <field name="content_lang2">
    Do not forsake wisdom, and she will protect you; love her, and she
    will watch over you. Wisdom is supreme; therefore get wisdom. Though
    it cost all you have, get understanding
  </field>
  <field name="content_lang3">
    N'abandonne pas la sagesse, et elle te gardera, aime-la, et elle te
    protégera. Voici le début de la sagesse: acquiers la sagesse,
    procure-toi le discernement au prix de tout ce que tu possèdes.
  </field>
</doc>
```



注意这个代码清单中有 4 份文档，一份只包含英语，一份只包含西班牙语，一份只包含法语，另一份同时包含这三种语种。代码清单 14.12 展示了将这些文档加载到 Solr 中时会发生什么。

代码清单 14.12 语种识别操作

发送文档

```
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/langid/update
-jar post.jar ch14/documents/langid.xml
```

查询请求

```
http://localhost:8983/solr/langid/select?
q=*&
fl=title,language,languages
```

搜索结果

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 0,
    "response": { "numFound": 4, "start": 0, "docs": [
      {
        "title": "The Adventures of Huckleberry Finn",
        "language": "en",
        "languages": ["en"]},
      {
        "title": "Les Misérables",
        "language": "fr",
        "languages": ["fr"]},
      {
        "title": "Don Quixote",
        "language": "es",
        "languages": ["es"]},
      {
        "title": "Proverbs",
        "language": "fr",
        "languages": ["fr",
          "en",
          "es"]}
    ]}
  }
}
```

←

←

←

←

语种字段包含文档被检测到的主要语种。

每个文档的语种字段可以包含多个被检测到的语种。

正如所预料的，Les Misérables 被定义为一个法语文档，Don Quixote 被定义为一个西班牙语文档，The Adventures of Huckleberry Finn 在主 language 字段和 (多值) languages 字段中都被定义为一个英语文档。以 Proverbs 为例，它同时被翻译成了上述三种语言，languages 字段正确地列出 en (English)、es (Spanish) 和 fr (French) 作为识别出的语种。单值 language 字段只列出了一种语言的代码，在本示例中是 fr，它是对这篇文档主要语种的最佳预测。

### 将动态内容映射到一个特定语种字段

向一个文档中添加一个语种字段很有趣，而且对一个特定语种进行语种分面或结果限制可能很有用。如果可以使用语种识别将内容映射到不同的分析器上，代码清单 14.12 中的语种识别实战对多语种搜索会有帮助。

对于一个 `field-per-language` 的设置，语种识别器和更新处理器作为有用的扩展功能，都是开箱即用的。表 14.4 显示了语种识别更新处理器中可获得的一些额外的可用语种识别选项，其中有几个可以支持基于语种识别将内容映射到不同的字段中。

表 14.4 Solr 的语种识别更新处理器中的可用参数

参数	说明	类型	默认值
<code>langid</code>	启用 / 禁用语种检测	布尔	<code>true</code>
<code>langid.fl</code>	必备：逗号或空格分隔的字段列表，用于语种检测	字符串	--
<code>langid.langField</code>	必备：指定首要识别语言的映射字段	字符串	--
<code>langid.langsField</code>	指定一个字段及阈值，大于 <code>langid.threshold</code> 的所有识别语种代码都映射到该字段	字符串	--
<code>langid.override</code>	若 <code>langField</code> 与 <code>langsField</code> 字段内容已包含取值，确定其是否可以被覆盖。若该字段取值为 <code>false</code> ，指定白名单 <code>whitelisted</code> ( <code>langid.whitelist</code> )， <code>langField</code> 之前的值会复制到 <code>langsField</code>	布尔	<code>false</code>
<code>langid.threshold</code>	指定语种识别可接受的得分必须介于 0 与 1 之间。对于长文本字段而言，高阈值（如 0.8）会产生更好的结果。对于短文本字段，可能需要降低阈值	浮点数	0.5
<code>langid.whitelist</code>	指定允许的语种识别代码列表。如果指定 <code>langid.map</code> ，则可以使用白名单来确保文档只索引 <code>schema</code> 中存在的字段	字符串	--
<code>langid.map</code>	启用字段名称映射。若为 <code>true</code> ，Solr 会将 <code>langid.fl</code> 指定的每个字段内容映射到新的语言字段。新字段格式为 <code>fieldname_xx</code> ，其中 <code>xx</code> 表示识别的语种代码	布尔	<code>false</code>
<code>langid.map.fl</code>	使用 <code>langid.map</code> 逗号分隔的字段列表，而不是 <code>langid.fl</code> 中的所有字段	字符串	--
<code>langid.map.keepOrig</code>	若为 <code>true</code> ，Solr 会保留原本字段的内容副本，另外的版本将移动到语种专指字段	布尔	<code>false</code>
<code>langid.map.individual</code>	若为 <code>true</code> ，Solr 会为每个字段进行单独检测与语种映射。如果每个字段使用不同语种，并希望这种这些语种，而不是全局文档语种，请设置此参数	布尔	<code>false</code>
<code>langid.map.individual.fl</code>	<code>langid.map.individual</code> 逗号分隔的字段列表与 <code>langid.fl</code> 中指定的字段不同。若设置此参数，只有存在于列表中的字段才会根据它们各自的语种进行映射。 <code>langid.fl</code> 的其他字段将使用全局文档语种进行映射	字符串	

续表

参数	说明	类型	默认值
langid. fallbackFields	指定一个或多个字段，按顺序检查回退语种值，防止没有检测到满足 langid.threshold 分数的语种，或检测到的语种不在 langid.whitelist 中。如果没有找到合适的回退语种，Solr 会使用 langid.fallback 中指定的语种代码	字符串	--
langid.fallback	如果未检测到语种，也没有在 langid.fallbackFields 中发现语种的情况下，指定一种语种代码。如果未定义回退，则最终的语种代码是一个空字符串，这可能会导致意外行为	字符串	--
langid.map.lcmap	允许将语种代码映射为自主选定的字段前缀。例如，ja: cjk zh:cjk ko:cjk 将日语、汉语与汉语映射到格式为 <fieldname>_cjk 字段	字符串	--
langid.map.pattern & langid.map. replace	允许覆盖为特定语言字段映射的默认字段名格式。默认情况下，字段映射到 <fieldname>_<language>。pattern 是 Java 正则表达式中的模式，replace 是 Java 正则表达式中的替换。<language> 值会替换成识别语种的映射扩展	Java 正则 表达式/ 替换	--
langid. enforceSchema	若为 false，更新处理器不会根据 schema 来验证字段名。如果要创建临时字段，并计划稍后重命名或删除的情况比较适用	布尔	true

截至目前，读者已经了解了基本的语种识别的执行方法（代码清单 14.12），表 14.4 展示了 Solr 的语种识别更新处理器提供的附加功能和它的灵活性。如果尝试用 14.6.1 小节中提到的 separate-field-per-language 方法来实现多语种搜索，将内容自动映射到特定语种字段中是很容易的，无须编码。代码清单 14.13 展示了如何配置表 14.4 中的一些选项来让 Solr 将内容自动映射到已识别的特定语种字段。

代码清单 14.13 将内容自动映射到特定语种字段

SolrConfig.xml

```
...
<updateRequestProcessorChain name="langid">
  <processor class="org.apache.solr.update.processor.
    LangDetectLanguageIdentifierUpdateProcessorFactory">
    <lst name="invariants">
      <str name="langid.fl">content</str>
      <str name="langid.langField">language</str>
      <str name="langid.map">true</str>
      <str name="langid.map.fl">content</str>
      <str name="langid.whitelist">en,es,fr</str>
      <str name="langid.map.lcmap">en:english es:spanish fr:french</str>
      <str name="langid.fallback">en</str>
    </lst>
  </processor>
```

```

...
</updateRequestProcessorChain>
...
<requestHandler name="/update" class="solr.UpdateRequestHandler">
  <lst name="invariants">
    <str name="update.chain">langid</str>
  </lst>
</requestHandler>
...

```

### Schema.xml

```

...
<field name="language" type="string" indexed="true" stored="true" />
...
<dynamicField name="*_english" type="text_english" indexed="true"
  stored="true" multiValued="true"/>
<dynamicField name="*_spanish" type="text_spanish" indexed="true"
  stored="true" multiValued="true"/>
<dynamicField name="*_french" type="text_french" indexed="true"
  stored="true" multiValued="true"/>
...

```

该代码清单表明，title 字段和 content 字段都可以用于识别文档中的语种，但只有 content 字段应该被映射到一个特定语种字段。Whitelist 字段中也被指定为仅包括三个语种：fr、en 和 es。

为了更清楚地展示，我们为这些语种创建了动态字段，它们根据语种识别被映射到 content\_french、content\_english 或 content\_spanish 字段。如果一定要使用 Solr 的示例 schema.xml 文件中的这三个语种类型的默认动态字段扩展，则不必提供自定义的 langid.map.lcmap 参数来为这些语种明确地映射扩展字段。代码清单 14.14 展示了上述语种检测的配置。

### 代码清单 14.14 语种检测将内容映射到特定语种字段

#### 发送文档

```

cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/langid2/update
  -jar post.jar ch14/documents/langid.xml

```

#### 查询请求

```

http://localhost:8983/solr/langid2/select?
q=id:[1 TO 3]&
fl=title,language,content_english,content_spanish,content_french&
defType=edismax&
qf=content_english content_spanish content_french

```

#### 搜索结果

```

{
  "responseHeader": {

```

每个文档的原始内容对应到一个语种字段。

对任意关键词查询使用 eDisMax 查询解析器。

每个文档  
的原始内容  
对应到一个  
语种  
字段。

```
"status":0,
"QTime":0},
"response":{"numFound":3,"start":0,"docs":[
  {
    "title":"The Adventures of Huckleberry Finn",
    "language":"en",
    "content_english":["YOU don't know about me without..."]},
  {
    "title":"Les Misérables",
    "language":"fr",
    "content_french":["Nul n'aurait pu le dire; tout ce..."]},
  {
    "title":"Don Quixote",
    "language":"es",
    "content_spanish":["Demasiada cordura puede ser la peor..."]}
]}
```

Solr 的语种识别器更新处理器链为 `field-per-language` 多语种方法提供了很好的自动识别语种类型的支持。在该示例中，每个与示例查询匹配的文档都是最初在 `content` 字段上构建索引的，然而当每个文档中的语种被识别后，内容将会被转移到对应的语种字段。正如 14.6.1 小节中讨论过的，通过使用 `eDisMax` 查询解析器进行跨字段搜索，可以将跨多语种检测和搜索的复杂性完全交给 Solr 中 `field-per-language` 字段的配置。如果考虑使用本节中提到的另外两种多语种搜索的方法，将内容重定向到一个单独的 Solr 内核来设置 `language-per-core`，这将会非常困难，因为内容要么必须被发送到每个内核，然后被不属于它们的内核丢弃，要么不得不在语种检测后从一个 Solr 内核转移到另一个 Solr 内核中——Solr 目前并不直接支持该操作。可以考虑使用支持 `multiple-languages-per-field` 的方法，该方法可以很容易地用一小段额外的代码完成。

## 14.7.2 在一个字段中动态分配语种检测分析器

识别语种并将内容映射到不同的字段以及识别语种并将内容映射（作为前缀）到相同的字段，这两者在本质上并没有太大的区别。它们都可以通过扩展 `LangDetectLanguageIdentifierUpdateProcessor` 并做一些简单的改变来实现。代码清单 14.15 展示了一个实现该目的的扩展更新处理器。

代码清单 14.15 `MultiTextField` 的 `LanguageIdentifierUpdateProcessor`

```
public class MultiTextLanguageIdentifierUpdateProcessor
    extends LangDetectLanguageIdentifierUpdateProcessor {

    private static String MULTI_TEXT_FIELD_LANGID = "mtf-langid";
    private static String PREPEND_GRANULARITY =
        MULTI_TEXT_FIELD_LANGID + ".prependGranularity";
```

```

private final static String HIDE_PREPENDED_LANGS =
    MULTI_TEXT_FIELD_LANGID + ".hidePrependedLangs";

private enum PrependGranularities { document, field, fieldValue }

private static String PREPEND_FIELDS =
    MULTI_TEXT_FIELD_LANGID + ".prependFields";

protected IndexSchema indexSchema;
protected Collection<String> prependFields = new LinkedHashSet<String>();
private PrependGranularities prependGranularity =
    PrependGranularities.document;
private Boolean hidePrependedLangs = true;

public MultiTextFieldLanguageIdentifierUpdateProcessor(SolrQueryRequest req,
    SolrQueryResponse rsp, UpdateRequestProcessor next) {
    super(req, rsp, next);
    indexSchema = req.getSchema();
    initParams(req.getParams());
}

private void initParams(SolrParams params) {
    //Default parameter initialization ... accompanying source code
}

@Override
protected SolrInputDocument process(SolrInputDocument doc) {

    SolrInputDocument outputDocument = super.process(doc);

    Collection<String> fieldNames = new ArrayList<String>();
    for (String nextFieldName : outputDocument.getFieldNames()) {
        fieldNames.add(nextFieldName);
    }

    List<DetectedLanguage> documentLangs =
        this.detectLanguage(this.concatFields(doc, this.inputFields));

    for (String nextFieldName : this.prependFields) {
        if (indexSchema.getFieldOrNull(nextFieldName) != null) {
            if (indexSchema.getField(nextFieldName).getType()
                instanceof MultiTextField) {
                outputDocument = detectAndPrependLanguages(
                    outputDocument, nextFieldName, documentLangs);
            } ...
        }
    }

    return outputDocument;
}

protected SolrInputDocument detectAndPrependLanguages(
    SolrInputDocument doc,
    String multiTextFieldName,
    List<DetectedLanguage> documentLangs){

    MultiTextField mtf = (MultiTextField) indexSchema

```

使用预备  
语种对所有  
字段进  
行标记。

在文档层次检  
测语种。

若发出请求，在字段层次检测语种。

检测每个字段值（包括多值字段）的语种。

```

        .getFieldType(multiTextFieldName);
MultiTextFieldAnalyzer mtfAnalyzer = (MultiTextFieldAnalyzer) mtf
        .getAnalyzer();

List<DetectedLanguage> fieldLangs = null;
if (this.prependGranularity == PrependGranularities.field
    || this.prependGranularity == PrependGranularities.fieldValue) {

    fieldLangs = this.detectLanguage(
        this.concatFields(doc, new String[] { multiTextFieldName }));
}

if (fieldLangs == null || fieldLangs.size() == 0) {
    fieldLangs = documentLangs;
}

SolrInputField inputField = doc.getField(multiTextFieldName);
SolrInputField outputField = new SolrInputField(inputField.getName());
if (inputField.getValues() != null) {
    for (final Object inputValue : inputField.getValues()) {
        Object outputValue = inputValue;
        List<DetectedLanguage> fieldValueLangs = null;
        if (this.prependGranularity == PrependGranularities.fieldValue) {
            if (inputValue instanceof String) {
                fieldValueLangs = this.detectLanguage(inputValue.toString());
            }
        }

        if (fieldValueLangs == null || fieldValueLangs.size() == 0) {
            fieldValueLangs = fieldLangs;
        }

        LinkedHashSet<String> langsToPrepend = new LinkedHashSet<String>();
        for (DetectedLanguage lang : fieldValueLangs) {
            langsToPrepend.add(lang.getLangCode());
        }

        StringBuilder fieldLangsPrefix = new StringBuilder();
        for (String lang : langsToPrepend) {
            if (mtfAnalyzer.Settings.ignoreMissingMappings
                || mtfAnalyzer.Settings.fieldMappings.containsKey(lang)
                || indexSchema.getFieldOrNull(lang) != null) {

                if (fieldLangsPrefix.length() > 0) {
                    fieldLangsPrefix.append(
                        mtfAnalyzer.Settings.multiKeyDelimiter);
                }

                fieldLangsPrefix.append(lang);
            }
        }

        if (fieldLangsPrefix.length() > 0) {
            fieldLangsPrefix.append(
                mtfAnalyzer.Settings.keyFromTextDelimiter);
        }
    }
}

```

为检测到的语种赋予前缀。



```

        if (this.hidePrependedLangs) {
            fieldLangsPrefix.insert(0, '[');
            fieldLangsPrefix.append(']');
        }

        outputValue = fieldLangsPrefix + (String) outputValue;
        outputField.addValue(outputValue, 1.0F);
    }
}

outputField.setBoost(inputField.getBoost());
doc.removeField(multiTextFieldName);
doc.put(multiTextFieldName, outputField);
return doc;
}

```

具体语法并未存储在预备语种中。

包含语种前缀的新字段会取代原始字段。

MultiTextFieldLanguageIdentifierUpdateProcessor 类继承自 LangDetectLanguageIdentifierUpdateProcessor 类，并为每个检测的语种的内容字段添加语种前缀。代码清单 14.16 展示了如何使用 MultiTextFieldLanguageIdentifierUpdateProcessorFactory 类（可在本章代码中下载）来配置更新处理器。

代码清单 14.16 在 solrconfig.xml 文件中配置语种识别器 MultiText

```

<requestHandler name="/update" class="solr.UpdateRequestHandler">
  <lst name="invariants">
    <str name="update.chain">multi-langid</str>
  </lst>
</requestHandler>

<updateRequestProcessorChain name="multi-langid">
  <processor class=
    "sia.ch14.MultiTextFieldLanguageIdentifierUpdateProcessorFactory">
    <lst name="invariants">
      <str name="langid.fl">title,content</str>
      <str name="langid.langField">language</str>
      <str name="langid.whitelist">en,es,fr</str>
      <str name="langid.fallback">en</str>
    </lst>
    <lst name="defaults">
      <str name="mtf-langid.prependFields">content</str>
      <str name="mtf-langid.prependGranularity">fieldValue</str>
      <str name="mtf-langid.hidePrependedLangs">false</str>
    </lst>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>

```

新支持的配置包括：指定自动预置已检测到的语种（mtf-langid.prependFields）字段的列表；通过设定 mtf-langid.prepend-Granularity 的参数

值为 document、field 或 fieldValue 来指定是否要在每个文档、每个字段或者每个字段值（对于多值字段而言）上进行语种检测；以及通过设定 mtf-langid.hidePrependLangs 的参数值为 false，来防止预置的语种出现在字段的存储版本中。代码清单 14.17 演示了如何使用代码清单 14.16 中的配置在 Solr 中索引示例文档。

#### 代码清单 14.17 使用语种识别器 MultiText 索引文档

##### 发送文档

```
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/multi-langid/update
  -jar post.jar chl4/documents/multi-langid.xml
```

##### 查询请求

```
http://localhost:8983/solr/multi-langid/select?
q=*&
df=content&
fl=title,content,language
```

##### 搜索结果

```
{
  "responseHeader":{
    "status":0,
    "QTime":0},
  "response":{"numFound":4,"start":0,"docs":[
    {
      "title":"The Adventures of Huckleberry Finn",
      "language":"en",
      "content":["en|YOU don't know about me without..."]},
    {
      "title":"Les Misérables",
      "language":"fr",
      "content":["fr|Nul n'aurait pu le dire; tout ce..."]},
    {
      "title":"Don Quixote",
      "language":"es",
      "content":["es|Demasiada cordura puede ser la peor..."]},
    {
      "title":"Proverbs",
      "language":"es",
      "content":[
        "es|No la abandones y ella velará sobre ti...",
        "en|Do not forsake wisdom, and she will protect you...",
        "fr|N'abandonne pas la sagesse, et elle te gardera..."]}]}
```

在字段中每一行都识别出语种 (prepend Granularity=fieldValue)。

每个字段值都标有语种前缀。

每个文档的 content 字段都被预置了该文本检测到的语种类型，这是因为 content 字段被定义为一个 MultiText 字段，而且代码清单 14.16 中指定了

content 字段需要有预置语种 (mtf-langid.prependFields)。此外, 需要注意在最后的文档中, 每项的语种都被识别了出来。因为要求预置的粒度是每个字段值, 而 content 字段是一个多值字段, 因而就要求字段中每个单独的项要有自己独立的语种识别器。已执行的查询是一个开放的搜索, 但也可以使用代码清单 14.17 中展示的语法来执行任何一个对 MultiTextField 字段的查询。

在代码清单 14.16 中的示例配置中没有用到的一个参数是 mtf-langid.hidePrependedLangs。这个参数可以从 MultiTextFields 字段要被索引的值中分离出已存入的值。通过将这个值设置为 true, content 字段将会继续把语种代码预置到字段的索引版本中 (以供 MultiTextField 处理), 但是语种代码将不会被添加到字段的存储版本中。代码清单 14.18 演示了在设置 mtf-langid.hidePrependedLangs 为 true 时, 向 Solr 中加载相同文档的示例。

代码清单 14.18 在不修改存储值的情况下预置已检测的语种类型

#### 发送文档

```
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/multi-langid/update
  -D?mtf-langid.hidePrependedLangs=true
  -jar post.jar ch14/documents/multi-langid.xml
```

修改默认配置,  
开启添加语种  
标识隐藏。

#### 查询请求

```
http://localhost:8983/solr/multi-langid/select?
q=en,es,fr|abandon&
df=content&
fl=title,content,language
```

语种前缀不存储, 但每  
种语言仍然会进行词干  
提取。

#### 搜索结果

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 0,
    "response": { "numFound": 4, "start": 0, "docs": [
      {
        "title": "Proverbs",
        "language": "es",
        "content": [
          "No la abandonas y ella velará sobre ti...",
          "Do not forsake wisdom, and she will protect you...",
          "N'abandonne pas la sagesse, et elle te gardera..."
        ]
      }
    ]
  }
}
```

语种前缀  
不存储,  
但每种语  
言仍然会  
进行词干  
提取。

虽然语种前缀没有被存储, 但是词干提取仍然在如期运行, 词 abandones (西班牙语) 和 abandonne (法语) 在索引时都被词干化为词 abandon。(可以在查询语句中添加参数 facet=true&facet.field=content 来检查每个被索引的词)。这可以实现两全其美的效果: 基于动态检测语种实现多语种文本分析, 而不需要根

据存储值以任何形式修改输入内容（在搜索结果中返回或者在高亮中使用）。实现存储一个不同的值（而不是存储将被索引的值）的具体细节相当复杂，并非本章内容的重点（如果读者十分好奇，可以自行探索相关代码）。读者现在只需要理解一点，即该功能将语种前缀包在方括号中，以这样方式将其应用在 `MultiTextField` 字段上。例如，如果输入 `[es|]no la abandones...`，而不是输入 `es|no la abandones...`（这样会索引并存储语种前缀），则 `MultiTextField` 字段只索引而不保存语种前缀。如果参考代码清单 14.15 中的代码，可以看到当 `mtflangid.hidePrependedLanguages` 被设置为 `true` 时，这个特殊的中括号会被插入。如果不使用自动语种检测，仍然可以自由地通过这个特殊中括号的语法来实现相同的效果。

将 Solr 中的内置语种处理和分析功能结合到一起的方法有很多，本章主要介绍了 Solr 中提供的方法，并在此基础上介绍了如何根据具体需求轻松地扩展这些方法。

## 14.8 本章小结

跨语种全文搜索是 Solr 的一大特色功能。无论是使用内置的 Lucene 词干提取器还是其他一些开源的或商业的词形还原库，在特定语种形式中找到单词的缩写可以大大提高应用程序搜索的查全率。本章涵盖了处理语种差异的常用技术，其中包括很多特定语种分词器（Tokenizer）和词素过滤器（Token Filter）的配置。同时还展示了处理多语种内容的三种方法：`field-per-language`、`core-per-language` 和 `multiple-languages-per-field`。虽然 `field-per-language` 方法是 Solr 中开箱即用的，但是随着查询字段数的增多，它可能会导致查询速度变慢，它将迫使在所有的查询中都使用 `DisMax` 查询解析器。当要处理的文档规模很大时，`core-per-language` 方法可以提供更快的查询速度，但是管理多个 Solr 内核带来了相当大的服务器管理开销，并且它不能很好地与 Solr 中的自动语种识别进行整合。

本章还展示了创建自定义的字段类型、分析器和分词器的方法，它们能动态处理每个字段中的多语种。这是三种方法中最灵活、扩展性最好的一种方法，但其实现很复杂，并且它要求为每个词都添加应被处理的语种列表前缀，因而导致查询语法很累赘。本章包含了一个文档语种识别的示例，包括将内容自动映射到特定的语种字段，使用代码将语种识别和 `multiple-languages-per-field` 方法整合在一起。现在，读者应该已经掌握了大量工具，可以实现很好的多语种搜索体验了。

# 15 复杂查询操作

## 本章要点

- 自定义函数查询
- 地理空间搜索
- 多层次分面透视
- 在查询中引用外部数据源
- 使用 Solr 做大数据分析

Solr 作为一个强大的文本搜索引擎，通过关键词查找和返回文档仅是最基础的功能。Solr 的很多核心的文档搜索功能（例如，丰富的文本分析、高亮、结果分组）增强了搜索结果的相关程度与有用性。对于很多搜索应用而言，呈现最佳匹配的文档是其最重要的功能，除此之外生成结果集报表、进一步支持数据分析也是 Solr 常见功能。Solr 的分面透视（pivot faceting）功能可以返回多层次的分面结果，计算出单个 Solr 请求中聚合点的任意数量，这将使得数据分析报表的制作更高效。

Solr 的另一个核心功能是，在查询时执行数据函数，用于过滤文档、提高相关度计算值、对结果排序以及将动态生成的内容追加到每个文档返回的字段中。Solr 还拥有强大的地理空间搜索功能，支持基于点和形状的多边形搜索，也支持基于纬度 / 经度坐标的地理半径搜索，并根据邻近程度来提高相关度和排序。

在查询时，文档集引用外部文档可能是有用或必要的。Solr 文档可以包含引用外部字段的字段，也可以在同一 Solr 实例中，在任意 Solr 内核的外键字段上执行基本的连接（join）操作。本章会依次演示这些复杂的数据操作，在传统文本搜索与

检索应用之上，展示 Solr 作为一个数据分析引擎的优点以及若干缺点。

## 15.1 函数查询

Solr 的函数可以动态计算每个文档的值，而不是返回在索引阶段对应字段的静态数值集。函数查询是一类特殊的查询，它可以像关键词一样添加到查询中，对所有文档进行匹配并返回它们的函数计算值作为文档得分。使用函数查询，函数计算结果将用于修改相关度得分或用于搜索结果的排序。在应用程序层，函数计算的结果可以作为一个动态字段添加到每个文档。函数也可以嵌套，即一个函数的输出可以作为另一个函数的输入，函数允许嵌套任意多层。

### 15.1.1 函数语法

Solr 的标准函数语法首先定义一个函数名，后面紧跟一对括号，括号中包括零个、一个或多个输入参数，参数之间以逗号分隔：

```
functionName()  
functionName(input1)  
functionName(input1, input2)  
functionName(input1, input2, ... inputN)
```

以下内容都可以作为函数的输入。

- 常量（数值或字符串常量）  
语法：100, 1.45, "hello world"
- 字段  
语法：fieldName, field(fieldName)
- 另一个函数  
语法：functionName(...)
- 替代参数  
语法：q={!func}min(\$f1,\$f2)&f1=sqrt(popularity)&f2=1

Solr 将文档中每个输入参数的类型定义为函数，初看可能会对此感到困惑。绝大多数函数遵循标准的函数语法，但常量函数、字段函数和替代参数是简化语法的特例。常量函数的语法就是常量值本身；字段函数的语法是字段的名称，可以选择性地在函数中包含 field 命名；替代参数的语法是 \$parameter，其表示 URL 请求的查询字符串参数。除此之外，其他函数都遵循标准的函数语法。

由于函数的所有输入可以看成函数本身（即使输入是一个常量函数），标准的函数语法可以在概念上简化为 functionName(function1, ... functionN)。假设文档中的 fieldContainingNumber 字段包含值 -99，思考以下函数：



<code>max(2, fieldContainingNumber)</code>	Result: 2
<code>max(fieldContainingNumber, 2)</code>	Result: 2
<code>max(2, -99)</code>	Result: 2
<code>max(-99, 2)</code>	Result: 2
<code>max(2, field(fieldContainingNumber))</code>	Result: 2
<code>max(field(fieldContainingNumber), add(1,1))</code>	Result: 2

不难看出，每个函数可以容易地将字段函数替换为常量函数或其他标准函数。虽然每个例子中计算输入参数的命令和方法不同，但都返回了 -99 和 2 之间的最大值。将一个函数输入作为另一个函数的好处是，以有趣的方式组合函数来实现复杂的计算。

并不是所有的函数都接受相同类型的输入参数。一些函数将常量值输入转变为字符串，另一些函数则将其转变为整数或者浮点数。假设 `fieldContainingString` 赋予 "hallo" 值，如下所示：

<code>strdist("hello", fieldContainingString, edit)</code>	Result: 0.8
<code>strdist("hello", "hallo", "edit")</code>	Result: 0.8

`strdist` 函数基于一种特殊的算法。（由第三个参数定义，在本例中常量函数是 "edit" 的文本值，即 `edit`）来计算两个字符串的相似度。如果在此函数中输入了错误的类型，会出现什么结果呢？

<code>strdist("hello", 1000, edit)</code>	Result: 0
<code>strdist(1000, "1000", edit)</code>	Result: 1
<code>strdist("1001", 1000, edit)</code>	Result: 0.75

你可能预计这个函数会抛出异常，但实际上它会将常量值转换成适合的类型，这里是将数值型常量 "1000" 转换成字符串 "1000"。很多时候这种转换是不可能的（例如，怎样确保字符串正确地转换为数值呢？）这种情况下，通常会收到 Solr 异常提醒。需要明确一点，虽然函数嵌套语法是通用的，但并不是所有的函数都可以组合成功。

### 启动本章预配置的例子

与先前章节一样，本章的示例文件已经预置在单独的 Solr 内核中，启动一次 Solr 就能操作本章所有例子。为了使 Solr 立即启动并运行本章节所有例子，请执行以下操作：

```
cd $SOLR_INSTALL/example/
cp -r $SOLR_IN_ACTION/example-docs/ch15/cores/Solr/
cp $SOLR_IN_ACTION/Solr-in-action.jar Solr/lib/
java -jar start.jar
```



Solr-in-action.jar 文件包含本章示例所需的绝大多数依赖项。另外，还有一个在 Java Topology Suite 上的外部依赖项需要额外安装，这会在后面讨论 Solr 地理搜索功能时介绍。

Solr 的函数通用性使得它们可以在 Solr 的各种核心功能上使用。函数可以影响相关度，可以过滤结果，可以用于排序，也可以对文档附加返回的函数值，甚至可以用在分面上。接下来的几个小节中将深入讨论函数的用法。

### 15.1.2 函数的搜索

在 Solr 中执行典型的关键词搜索时，每个关键词会在倒排文档中查找一遍，通过计算相关度得分来决定每个文档与关键词的匹配程度（详见第 3 章搜索过程的讲解）。查询并不局限于词项本身，也可以在查询中插入函数，将其视为另一个关键词。为了说明这一点，我们将示例文档索引到 Solr 内核 news 中：

```
cd $SOLR_IN_ACTION/example-docs/  
java -Durl=http://localhost:8983/solr/news/update  
-jar post.jar ch15/documents/news.xml
```

文档索引好之后，接下来介绍用户查询关键词中包含函数的情况：

```
http://localhost:8983/solr/news/select?  
q="United States" AND France AND President AND  
_val_:"recip(ord(date),1,100,100)"
```

该查询执行布尔搜索的关键词为 "United States"、France 和 President，以及一个返回值为 1~100 区间值的函数，这个函数用来衡量匹配文档的新旧程度（文档越新，返回值越高）。此查询有如下三个方面需要特别注意。

1. 语法 `_val_:"value"` 用来将一个查询函数（嵌套 `recip` 和 `ord` 函数，将在 15.1.5 节介绍）作为一个词项插入到用户主查询语句中。
2. 函数查询默认匹配所有文档。在前面的例子中，查询被限制在包含 "United States"、France 和 President 三个词项的所有文档中，函数查询作为额外的词项并没有改变查询匹配的文档结果数。
3. 一个查询的相关度评分是查询中每个词项相关度得分的总和。"United States"、France 和 President 三个词项都会得到各自基于 `tf-idf` 相似度计算（参见 3.2 节）的相关度得分，然而函数查询的得分是函数自身的取值。

基于这三点，示例函数查询的目的是让越新的文档相关度得分越高。具体而言，最新文档的相关度得分将获得 100 的加分，最旧文档的相关度得分将获得 1 的加分，其余文档根据其新旧程度获得 1~100 之间的某个加分。注意，每个文档的最后得分

经过规范化处理, 因此不会看到实际的 100 分加到每个文档的最后得分中, 只会看到越新的文档排名提升越多。如果从示例查询中移除函数, Solr 的搜索结果排序会发生变化。

除了根据文档的新旧程度之外, 还可以根据地理距离、流行度或者其他的一些计算指标来调整相关度。第 16 章将详细介绍相关度提升的一些方法。

### 在查询中挂接函数

函数在 Solr 中很常见, 它们可以用于提升查询 (参数 `q`) 的相关度, 或者在不同的查询解析器中调整特定的参数 (例如, 查询解析器 `eDisMax` 的参数 `bf`), 或者用作过滤器应用的一部分, 还可以用来排序文档, 以及返回文档的动态计算值等, 这些内容会在本章及下一章涉及。现在理解一个函数在查询中如何被调用很有必要。

本节中介绍的 `_val_:"functionName(...)"` 语法, 可以像关键词那样插入到查询的任何位置。第 7 章曾提到, Solr 包含一个函数查询解析器, 通过本地参数 `{!func}functionName(...)` 进行调用。两种方案可以实现相同的功能: 将函数的值作为一个词项添加到查询中, 它的相关度得分就是函数本身的价值。因此, 以下语句是等价的。

- 值挂接:

```
q=Solr AND _val_:"add(1, boostField)"
```

- 查询解析器挂接 (显式嵌套查询):

```
q=Solr AND _query_:"{!func}add(1, boostField)"
```

- 查询解析器挂接 (隐式嵌套查询):

```
q=Solr AND {!func v="add(1, boostField)"}
```

每个语句的输出结果都是一样的, 这里把几种形式都列出来, 以便于理解本章、下一章及其他 Solr 示例和文档中出现的这些语句。

通过向查询添加函数, 可以调整与查询匹配的文档相关度得分, 这个做法似乎很有用。事实上, 如果想要通过函数计算来过滤某个适合结果区间以外的结果, 函数查询就不那么有效了。所幸, Solr 提供了函数区间查询解析器来解决此类需求。

### frange 查询解析器

如果需要对搜索结果进行过滤, 只留下函数计算产生特定值的文档, 可以选择函数区间解析器 (Function Range query parser, 简称 `frange`)。frange 过滤器执行一个特定的函数查询, 然后过滤掉函数值落在最低值和最高值范围之外的文档。为了演示区间过滤功能, 我们使用一个函数来计算产品的营业税。首先, 需要向

Solr 内核 salestax 添加文档：

```
java -Durl=http://localhost:8983/solr/salestax/update  
-jar post.jar ch15/documents/salestax.xml
```

下面的查询会启动 frange 过滤器：

```
http://localhost:8983/solr/salestax/select?q=*&  
fq={!frange l=10 u=15}product(basePrice, sum(1, $userSalesTax))&  
userSalesTax=0.07
```

此查询通过 userSalesTax 参数来计算每件产品的总价格，同时将营业税添加到每件产品基础价格定义的 basePrice 字段。frange 查询解析器过滤了总价格在 10~15 区间以外的那些文档，上限和下限通过本地参数 l（最低）和 u（最高）来定义。上限和下限是默认的，如果只想匹配包含特定值的文档，可以将 l 和 u 的值设为同一个值。另外，上限值和下限值的设置是可选的，没有强制要求同时设置上下限。如有需要，frange 查询中的本地参数 incll（包含下限）和 inclu（包含上限）可设置为 false，这样可以过滤出不在区间范围内的文档。

匹配任意复杂函数的文档限定功能为自定义查询提供了灵活的工具实现。15.1.6 节将演示如何编写程序来实现将用户自定义的函数添加至 Solr，最大限度地在 Solr 中操作数据。现在我们已经了解了搜索的函数功能，并理解了函数值的计算方法，下面来看如何将动态计算出的函数值赋予静态字段。

### 15.1.3 以字段形式返回函数

15.1.1 小节介绍了所有的函数输入，包括常量和字段，在函数查询语法中都可视为函数本身。既然如此，函数和字段最终都会返回一个值，因此在 Solr 中其他地方中用函数替代字段也是可行的。

事实上，不仅可以计算每篇文档对应的函数值，也可以将文档的计算值当作伪字段（pseudo-field）返回。参见 15.1.2 节的营业税例子，我们提交一个请求，计算每篇文档中产品的总价格，然后和其他字段一起返回：

```
http://localhost:8983/solr/salestax/select?q=*&  
userSalesTax=0.07&  
fl=id,basePrice,product(basePrice, sum(1, $userSalesTax))
```

这个请求的搜索结果是怎样的？对于仅包含 ID 和 basePrice 字段的文档而言，搜索结果如下所示：

```
{
  "id": "1",
  "basePrice": 10.0,
  "product(basePrice, sum(1, $userSalesTax))": 10.700001
}
```

如你所见，向字段列表请求增加一个函数，会将一个新的字段添加到文档中。这并不是存储在索引中的真实字段，但会像存储字段一样返回到文档。

最后一个例子不好的地方在于，返回到文档中的伪字段名称是计算函数值的语法，调用 Solr 的应用程序要将其解析和映射为一个有意义的名称非常困难。幸好，Solr 允许为返回值的伪字段名称自定义别名，如下代码清单 15.1 所示。

### 代码清单 15.1 在伪字段中返回函数计算值

#### 查询请求

```
http://localhost:8983/solr/salestax/select?q=*:*&
  userSalesTax=0.07&
  fl=id,basePrice,totalPrice:product(basePrice, sum(1, $userSalesTax))
```

#### 查询结果

```
{
  {
    "id": "1",
    "basePrice": 10.0,
    "totalPrice": 10.700001
  }
}
```

请求一个伪字段用于计算基本价格和营业税之和

新的 totalPrice 字段包含在返回的文档中

动态添加的伪字段名称 totalPrice 使用语法 `totalPrice:product(basePrice, sum(1, $userSalesTax))` 来定义。冒号之前是伪字段的名称，冒号之后是计算伪字段值的函数。这让伪字段像真实字段一样返回函数值。事实上，动态计算的伪字段也可以覆盖一个真实字段。你可以想象这样的用例需要在不同用例中的同一个字段上返回不同的值，例如，基于用户访问权限清空字段，或为不同市场提供各种版本的内容翻译来修改字段值。在返回搜索结果之前，函数可以操作任何字段的取值。函数不仅可以修改返回的文档字段，还可以改变返回文档的排序。

### 15.1.4 函数排序

上一节提到，函数的计算结果可以作为一个字段添加到文档并返回到搜索结果。这是因为 ValueSource（查询函数）产生了 DocValues（每个文档与它们取值的映射），包含了每个文档的计算值。到这里我们就了解了如何根据每个文档计算出的函数值来过滤搜索结果集、修改文档的相关度，以及添加或修改搜索请求的返回字段和取值。接下来介绍如何根据计算出的函数值对搜索结果进行排序。

函数的排序语法与字段的排序语法的唯一不同之处在于，整个函数语法（引用参数包含全函数语法）取代了字段名称：

```
http://localhost:8983/solr/salestax/select?q=*&
  userSalesTax=0.07&
  sort=product(basePrice, sum(1, $userSalesTax)) asc, score desc
```

这个请求将根据之前计算的总价格来排序（从低到高），如果价格相同，则按照文档得分从高到低排序。当然，你可以将之前描述的所有方法组合起来构造更加复杂的查询：

```
http://localhost:8983/solr/salestax/select?
  q=_query_:"{!func}recip(ord(date),1,100,100)"&
  userSalesTax=0.07&
  totalPriceFunc=product(basePrice, sum(1, $userSalesTax))&
  fq={!frange l=10 u=15 v=$totalPriceFunc}&
  fl=*,totalPrice:$totalPriceFunc&
  sort=$totalPriceFunc asc, score desc
```

这个查询使用了解引用参数（独立出来的变量）`userSalesTax`，将它传递给间接引用的 `totalPriceFunc` 参数，然后在返回字段列表的过滤器（使用 `frange` 查询解析器）中再次使用 `totalPriceFunc` 参数，按照升序进行第一道排序。由于在查询中定义了一个单独的函数来提升新近文档的排名，第二道排序将按照相关度得分对相同价格的文档进行降序排序。

至此，搜索应用所需的一些函数工具都已介绍完毕。Solr 还有许多可供使用的函数，以下小节进行简单介绍。

### 15.1.5 Solr 的可用函数集

迄今为止，我们已经通过一些具体示例了解了 Solr 中的几种函数的使用方法。由于 Solr 提供的函数种类非常多，并且在持续增长，因此本章不会涉及所有的函数语法，有兴趣的话可以查访问 Solr Wiki 上最新的函数列表（<http://wiki.apache.org/solr/FunctionQuery>）。本节概览性地介绍 Solr 的函数类型，大体上了解有哪些可用的函数。Solr 的函数主要分为 4 类：数据转换函数、数学函数、相关度函数和布尔函数。

#### 数据转换函数

Solr 中最常用的函数是将数据从一种格式转换成另一种格式的函数。例如，函数 `map(x, min, max, target)`，如果 `x` 落在最小值与最大值之间，则会用 `x` 替换 `target` 值。如果 `price` 字段被设定为只包含比 10.00 大的值，则可以使用函数 `map(price, 0, 10, 10)` 将所有在 0 到 10.00 区间的值映射为最小值 10.00。



表 15.1 列出了一些其他有用的数据转换函数。

表 15.1 Solr 的数据转换函数

函数语法	说明
<code>def(x, y)</code>	若 $x$ 存在, 则返回 $x$ 的值, 否则返回 $y$ 的值
<code>field(fieldName)</code>	返回已索引字段的字段值, 包含该字段的每个文档最多一个值
<code>map(x, min, max, target)</code> <code>map(x, min, max, target, else)</code>	如果 $x$ 落在最小 $min$ 与最大 $max$ 之间, 则返回 $target$ 。 如果指定 (可选), 则 $else$ 返回其他
<code>ms(time2, time1)</code> <code>ms(time1)</code> <code>ms()</code>	返回 $time2 - time1$ 。若 $time2$ 未指定, 则为当前时间 NOW。 若 $time1$ 未指定, 则为 UNIX 系统纪元时间 (1/1/1970)
<code>ord(fieldName)</code>	返回搜索索引中词项的位置, 每个文档的每个字段有一个词项, 从 1 到索引唯一词项数
<code>rord(fieldName)</code>	返回 <code>ord</code> 函数的逆序
<code>scale(x, number1, number2)</code>	基于所有文档中 $x$ 的最大值与最小值, 介于 $number1$ 与 $number2$ 之间, 对每个文档的 $x$ 值进行缩放
<code>top(x)</code>	计算值来自顶层的 <code>IndexReader</code> , 而不是每个片段的 <code>IndexReaders</code> 。对需要考虑从其他文档取值的函数很有用。 例如, <code>ord</code> 与 <code>rord</code> 函数隐含使用 <code>top</code> 函数

## 数学函数

Solr 最常用的一些数据分析操作涉及数学函数。除了最常用的基本运算(加、减、乘、除), Solr 还支持各种数学运算, 甚至包括三角函数。表 15.2 列举了 Solr 的数学函数。

表 15.2 Solr 的数学函数

函数语法	说明
<code>abs(x)</code>	$x$ 的绝对值
<code>acos(x)</code>	$x$ 的反余弦值
<code>asin(x)</code>	$x$ 的正弦值
<code>atan(x)</code>	$x$ 的反正切值
<code>atan2(x, y)</code>	返回直角坐标 $(x, y)$ 转换为极坐标的方位角
<code>cbrt(x)</code>	$x$ 的立方根
<code>ceil(x)</code>	向上取整, 返回大于或等于 $x$ 的最接近整数
<code>cos(x)</code>	$x$ 的余弦值
<code>cosh(x)</code>	$x$ 的双曲线余弦值

续表

函数语法	说明
<code>deg(x)</code>	将 x 的弧度转换为度数
<code>div(x,y)</code>	x 除以 y
<code>e()</code>	返回欧拉数的近似值，以自然常数 e 为底
<code>exp(x)</code>	以自然常数 e 为底的指数函数，即 e 的 x 次方
<code>floor(x)</code>	向下取整，返回小于或等于 x 的最接近整数
<code>hypo(x,y)</code>	<code>sqrt(x<sup>2</sup>+y<sup>2</sup>)</code> 返回直角三角形的斜边长
<code>linear(m,x,b)</code>	<code>f(x) = m*x + b</code> 返回线性函数的值
<code>ln(x)</code>	x 的自然对数
<code>log(x)</code>	底数为 10 的 x 的对数
<code>pi()</code>	pi 是圆的周长与直径的比值
<code>pow(x,y)</code>	x 的 y 次方
<code>product(x,...n)</code>	将所有相乘，即求积
<code>mul(x,...n)</code>	
<code>rad(x)</code>	将 x 度转换为弧度
<code>recip(x,m,a,b)</code>	<code>a/(m*x+b)</code> 互逆函数实现
<code>rint(x)</code>	将 x 舍入为最接近的整数
<code>sin(x)</code>	x 的正弦值，以弧度表示
<code>sinh(x)</code>	x 的双曲正弦值
<code>sqrt(x)</code>	x 的平方根
<code>sub(x,y)</code>	x 减 y
<code>sum(x,...n)</code>	全部求和
<code>add(x,...n)</code>	
<code>tan(x)</code>	x 的正切值
<code>tanh(x)</code>	x 的双曲正切值

相关度函数

Solr 的相关度得分默认使用 `DefaultSimilarity` 类来计算，具体参见 3.2 节。这个类使用了来自搜索索引及查询术语的多种统计数据，以便识别出与查询最佳匹配的文档。虽然这些相关度统计数据通常仅用于计算每个文档的复合相关度得分，Solr 的相关度函数可以返回单独的统计数据供选用。所有关键的相关度统计数据都包含在内，例如，`tf-idf`。表 15.3 列出了 Solr 的相关度函数。



表 15.3 Solr 的相关度函数

函数语法	说明
<code>docfreq(fieldName, term)</code>	包含 <code>fieldName</code> 字段词项的文档数
<code>idf(fieldName, value)</code>	<code>fieldName</code> 字段值的反向文档频次计算
<code>maxdoc()</code>	索引中的文档数, 包含尚未清理的删除文档
<code>norm(fieldName)</code>	<code>fieldName</code> 字段存储在索引中的范数
<code>numdocs()</code>	索引中的文档数, 不包含尚未清理的删除文档
<code>query(subquery, defaultScore)</code>	与 <code>subquery</code> 匹配的文档得分, <code>defaultScore</code> 是不匹配 <code>subquery</code> 的文档得分
<code>sumtotaltermfreq(field)</code> <code>sttf(field)</code>	索引中该字段被索引的分词数
<code>termfreq(fieldName, term)</code>	词项在文档中该字段出现的次数
<code>tf(fieldName, term)</code>	字段中词项的 <code>tf</code> 因子是该字段的相似度
<code>totaltermfreq(fieldName, term)</code>	词项在整个索引中出现的次数
<code>ttf(fieldName, term)</code>	

从表 15.3 的相关度函数列表可以看出, 在查询中使用函数替代来重新计算相关度得分, 这是可以做到的, Solr 请求举例如下所示:

```
/select?
fq={!cache=false}content:"microsoft office"&
q={!func}sum(
  product(
    tf(content, "microsoft"),
    idf(content, "microsoft")
  ),
  product(
    tf(content, "office"),
    idf(content, "office")
  )
)
```

这个查询会过滤短语 "microsoft office", 并继续为 "Microsoft" 和 "office" 两个词项计算 `tf-idf` 值。由于函数在主查询中调用, 函数计算值返回为查询的相关度值。虽然这个例子不能代表全部默认的相关度计算, 通过它可以看到以各种方式重用相关度计算的元素, 在查询阶段有效地创建新的相关度计算, 是很容易做到的。Solr 提供的这些相关度函数为评分模型的构建提供了自由的发挥空间。

距离函数

有时，测量两个值之间的距离很有用，可能是地球上的两个点之间的地理距离，也可能是两个点或者向量之间的几何距离，甚至可能是两个字符串之间的字符距离，表 15.4 列出了 Solr 的距离函数。

表 15.4 Solr 的距离函数

函数语法	说明
<code>dist(power, x1, ... n1, x2 ... n2)</code>	根据 power 定义的距离度量，计算 2 个向量 / 点之间的距离。 最常见的 power 值包括： 0—稀疏计算 1—曼哈顿距离 2—欧几里得距离 无穷—无限范数（向量中的最大值）
<code>sqedist(x1, ... n1, x2 ... n2)</code>	<code>dist(2, ...)</code> 函数返回欧几里得距离的平方。如果只需要值的相对排序或相关度调整，而不需要精确的距离计算的话，这个函数会更有效，它消除了平方根计算
<code>hsin(radiusInKM, isDegrees, x1, y1, x2, y2)</code>	沿着球体计算半正矢，或更大的圆、距离，即球面上两点间的距离
<code>geohash(lat, lon)</code>	计算经纬度的 geohash 值。geohash 是地理位置的特殊字符串编码。该函数可以为 ghhsin 函数提供输入参数
<code>ghhsin(radiusInKM, geohash1, geohash2)</code>	对两个 geohash 值（不是度数或弧度）使用半正矢函数。geohash1 和 geohas2 字段可以是 geohash 字段（GeoHashField 类型）或来自 geohash 函数的计算结果
<code>strdist(s1, s2, distType)</code> <code>strdist(s1, s2, "ngram", ngramSize)</code>	计算两个字符串的字符相似度或间隔距离，介于 0（不相似）与 1（完全相同）之间。distType 的有效值为： jw—Jaro-Winkler 距离 edit—编辑距离（Levenshtein 距离） ngram—NGramDistance（编辑距离的 ngram 版本） 此外，实现 StringDistance 接口的任意类可以指定完整的限定名。如果编写自己的插件，这是有用的
<code>geodist(sfield, lat, lon)</code> <code>geodist(sfield, pt)</code> <code>geodist()</code>	返回地球上两点之间的距离，一个通过空间字段（sfield）指定，另一个通过坐标指定

如上表所示，Solr 支持许多距离函数。dist 函数允许用户定义 0-norm（稀疏计算）、1-norm（曼哈顿距离）、2-norm（欧式距离）或无穷范数（向量最大值）来计算在  $n$  维空间上的两个向量或点之间的距离。坐标数目必须为偶数，前一半的参数定义第一个点，后一半的参数定义第二个点。例如，二维空间里两点之间的欧式距离表示为 `dist(2, x1, y1, x2, y2)`，在三维空间里两点之间的曼哈顿距离表示

为 `dist(1, x1, y1, z1, x2, y2, z2)`。

`sqedist` 函数比欧式距离函数 (`dist` 函数的 2-norm) 的花销少, 它返回欧式距离的平方。平方后的欧式距离是二维空间里勾股定理 ( $a^2 + b^2 = c^2$ ) 的  $c^2$  部分。因为欧式距离必须对  $c^2$  开方来算出  $c$  值, 所以如果仅需要文档的相对顺序 (例如, 为了排序或相关度得分), 不需要返回距离值, 那么使用 `sqedist` 函数会让这个过程更加高效。

`hsin` 函数计算球体 (包括但不仅限于地球) 表面上两个点之间的距离。`hsin` 函数的 `radiusInKm` 参数是球体的半径。如果想计算地球上两点之间的距离 (这可能是最常见的情况), 则地球半径的近似值为 6371.01 (赤道处的半径) (由于地球不是一个完美球体, 位置精确度存在 0.5% 的偏差)。若指定了经 / 纬度, 则应该将 `isDegrees` 参数设置为 `true`; 若指定了点的弧度, 则应该将 `isDegrees` 参数设置为 `false`。`x1, y1` 的值定义第一个点, `x2, y2` 的值定义第二个点。

`ghhsin` 函数是 `hsin` 函数的一个版本, 它可以接受 `geohash` 值而不是角度或弧度。`geohash` 函数可以读取纬度和经度, 并将它们转换为地理散列编码值, 继而作为 `ghhsin` 函数的输入。如果搜索索引中有一个字段是存储地理散列编码值, 则需要使用这些函数。

`strdist` 函数通过比较字符差异性来确定两个字符串之间的距离, 主要用于相似词的模糊匹配。如果将一个字符串看成是包含多个字符的向量, `strdist` 函数可以计算出两个字符串 (字符向量) 之间的距离。相似度的计算结果会落在 0 (一点都不相似) 到 1 (几乎一样) 的区间。输入 `strdist` 函数的字符串输入习惯用 `s1` 和 `s2` 表示, `distType` 参数用于距离度量, 衡量两个字符的相似程度。如果指定为 "ngram" `distType`, 将默认使用两个字符来比较距离, 但可以通过传递可选的 `ngramSize` 参数来覆盖。

`geodist` 函数返回地球上两个点之间的距离。`sfield` (空间字段) 是一个 `LatLonType` 字段, 返回的距离是 `sfield` 的点与逗号分割的纬度和经度构成的点之间的公里数。如果调用 `geodist()` 函数时忘记了指定 `sfield` 和 `pt` 参数, 该函数会在请求中要求指定 `sfield` 和 `pt` 参数。`geodist` 函数在底层使用 `HaversineFunction` (`hsin`), 也假定了地球的半径, 使用一个 `LatLonType` 字段 (而不是分别存储每个值), 并提供更为简单的语法。`geodist` 是 Solr 中最常见的距离计算函数, 15.2 节会深入介绍地理空间搜索。

## 布尔函数

布尔运算不仅仅限于关键词查询, 还可以组合成任意复杂的函数查询。通过组合 `if`、`and`、`or`、`not`、`xor` 和 `exists` 函数可以检查字段值或者文档中的函数值, 并根据检查结果有条件地返回值。Solr 的布尔函数如表 15.5 所示。

表 15.5 Solr 的布尔函数

函数语法	说明
and(x, y)	若 x 与 y 都为 true，则返回 true
exists(x)	若 x 有值存在，则返回 true
if(x, trueValue, falseValue)	x 有 trueValue 与 falseValue 两个值，若 x 为 true，则返回 trueValue
not(x)	若 x 为 true，则返回 false。若 x 为 false，则返回 true
or(x, y)	若 x 为 true、y 为 true，或 x 与 y 都为 true，则返回 true；否则返回 false
xor(x, y)	若 x 或 y 为 true，则返回 true；若 x 和 y 都为 true，则返回 false

有如此丰富的（而且仍在不断丰富中）、开箱即用的数据转换函数、数学函数、相关度函数和布尔函数，我们就可以处理任何其他文档级别的计算，满足基于 Solr 的搜索应用。与 Solr 的其他许多功能类似，通过编写插件并实现自定义的函数可以很容易地扩展 Solr 的函数支持。下一小节将演示创建自定义的插件函数是多么轻而易举的事情。

15.1.6 自定义函数

对于你想执行的某些数据操作，Solr 中可能没有相应的内置函数。别担心，Solr 的用户自定义函数功能十分简便。代码在函数中执行，这意味着可以做很多技术处理，简单的如内存计算到链接外部文件，复杂的如从数据源导入更多信息，甚至执行一段任意代码等。自定义函数唯一的局限是等待函数计算完成的时间。因为函数代码在每个匹配文档中执行，它需要快速执行以便在合理的时间内进行搜索响应。

本节介绍基础函数的创建步骤，将多个字段值连接成一个字符串。创建 Solr 的自定义插件需要完成以下三个步骤：

- 1. 编写一个函数类。这个类继承 ValueSource 类，保证在搜索索引中的每个文档都返回一个计算值。
- 2. 编写 ValueSourceParser 类，它可以理解自定义函数的语法，并将它解析成第 1 步自定义的 ValueSource 函数需要的变量。
- 3. 向 Solrconfig.xml 文件添加一个 XML 元素，定义自定义函数的名称及 ValueSourceParser 的位置。当自定义函数通过函数名调用时，ValueSourceParser 类（来自第 2 步）将会解析 ValueSource（来自第 1 步）中的输入。

## 通过扩展 ValueSource 实现自定义函数

自定义的连接函数会扩展 Solr 的 ValueSource 类。ValueSource 通过 getValues 方法返回一个 FunctionValues 对象。FunctionValues 对象返回 Solr 索引中自定义函数任意文档的计算值。代码清单 15.2 演示了如何创建 ConcatenateFunction 类。

代码清单 15.2 连接两个值的自定义函数

```
public class ConcatenateFunction extends ValueSource {
    protected final ValueSource valueSource1;
    protected final ValueSource valueSource2;
    protected final String delimiter;

    public ConcatenateFunction(ValueSource valueSource1,
                               ValueSource valueSource2,
                               String delimiter) {

        if (valueSource1 == null || valueSource2 == null) {
            throw new SolrException(
                SolrException.ErrorCode.BAD_REQUEST,
                "One or more inputs missing for concatenate function"
            );
        }

        this.valueSource1 = valueSource1;
        this.valueSource2 = valueSource2;
        if (delimiter != null) {
            this.delimiter = delimiter;
        }
        else {
            this.delimiter = "";
        }
    }

    @Override
    public FunctionValues getValues(Map context,
                                    AtomicReaderContext readerContext) throws IOException {
        final FunctionValues firstValues = valueSource1.getValues(
            context, readerContext);
        final FunctionValues secondValues = valueSource2.getValues(
            context, readerContext);

        return new StrDocValues(this) {

            @Override
            public String strVal(int doc) {
                return firstValues.strVal(doc)
                    .concat(delimiter)
                    .concat(secondValues.strVal(doc));
            }

            @Override
            public String toString(int doc) {
```

1 请求两个 ValueSource 输入（字段、函数、值）。

2 可选的字符串分隔符。

3 该方法的函数返回一个 FunctionValues 对象。

4 字符串输出的特定 FunctionValues 对象。

5 对于任意文档 ID，返回函数计算值。

```

        StringBuilder sb = new StringBuilder();
        sb.append("concatenate(");
        sb.append("\"" + firstValues.toString(doc) + "\"")
            .append(',')
            .append("\"" + secondValues.toString(doc) + "\"")
            .append(',')
            .append("\"" + delimiter + "\"");
        sb.append(')');
        return sb.toString();
    }
};

@Override
public boolean equals(Object o) {
    if (this.getClass() != o.getClass()) return false;
    ConcatenateFunction other = (ConcatenateFunction) o;
    return this.valueSource1.equals(other.valueSource1)
        && this.valueSource2.equals(other.valueSource2)
        && this.delimiter == other.delimiter;
}

@Override
public int hashCode() {
    long combinedHashes;
    combinedHashes = (this.valueSource1.hashCode()
        + this.valueSource2.hashCode()
        + this.delimiter.hashCode());
    return (int) (combinedHashes ^ (combinedHashes >>> 32));
}

@Override
public String description() {
    return
        "Concatenates two values together with an optional delimiter";
}
}

```

对于 ConcatenateFunction 类需要特别注意的是它的输入参数和 getValues 函数返回的内容。就输入参数而言, ConcatenateFunction 类接受两个 ValueSource 对象 ❶ 和一个代表定界符 ❷ 的字符串。参见 15.1.1 节, 一个函数可作为另一个函数的输入。通过定义两个输入并将其连接成 ValueSource 对象 (而不是一个字符串或字段) 自定义的连接函数能够接受两个输入中的任何一个, 并能使用它们的函数值。虽然所有的输入参数都定义为 ValueSource 对象会带来高度灵活性, 但是需要注意在 ConcatenateFunction 构造器的第三个参数, 即分界符被定义为了字符串。这个输入参数也可以定义为 ValueSource, 从其他字段或者函数计算中取值。在这种情况下, 我们已经假定了分界符会在查询请求中显式传递。否则, 如果用户想要使用另一个字段或函数计算去动态发现分界符, 就需

要多次调用连接函数。

要理解 ConcatenateFunction 类的输出，需要掌握它的 `getValues` 方法 ❸。这个方法必须返回一个 `FunctionValues` 对象，但是连接操作的输出是一个字符串，系统内部将使用 `StrDocValues` 类 ❹。`StrDocValues` 类是 `FunctionValues` 的一个具体实现，返回的是字符串，而非整数型、布尔型或其他类型。`FunctionValues` 有许多子类，其中一些带有特定的缓存功能，通过调整内存使用量提高运行速度。如果考虑到性能优化，就需要进一步查看 Solr 代码库中的这些子类。

`StrDocValues` 对象 ❺ 包含 `strVal(docid)` 方法，调用该函数时，文档集中每个文档都要用此方法一次。这个方法计算函数的返回值，对于性能消耗较大的查询，由于搜索索引的每个文档都会调用此方法，确保每个方法的查询速度尽量达到最快是最重要的。

### 通过 ValueSourceParser 解析函数输入

了解了函数返回值的计算方法，接下来将介绍如何将请求输入解析到 `ConcatenateFunction` 对象。代码清单 15.3 演示了如何将输入解析到函数中。

代码清单 15.3 将查询请求的输入参数映射到函数

```
public class ConcatenateFunctionParser extends ValueSourceParser {
    public ValueSource parse(FunctionQParser fqp) throws SyntaxError {
        ValueSource value1 = fqp.parseValueSource();
        ValueSource value2 = fqp.parseValueSource();
        String delimiter = null;

        if (fqp.hasMoreArguments()) {
            delimiter = fqp.parseArg();
        }

        return new ConcatenateFunction(value1, value2, delimiter);
    }
}
```

将输入作为 ValueSources 进行解析，这个类型可以表示其他嵌套函数。

将分隔符解析为显式的字符串值。

以上代码验证了使用 `FunctionQParser` 对象可以很容易地将输入值解析到用户自定义的函数。`FunctionQParser` 解析标准的函数语法是 `functionName(input1,input2,...)`，基于请求函数的名称来选择合适的 `ValueSourceParser`。内部而言，`FunctionQParser` 拥有所有的函数输入，我们可以通过调用多种解析方法中的一个，如 `parseValueSource()`、`parseArg()` 或者 `parseFloat()`，从而获得函数的输入内容。

在 `ConcatenateFunctionParser` 的例子中，我们预设了两个 `ValueSource` 输入（字段、用户输入字符串，或者其他函数）和一个可选的字符串分界符。



ConcatenateFunctionParser 读取查询请求中的输入后, 创建 Concatenate-Function 对象并导入输入内容。

自定义函数调用

创建好自定义连接函数所需要的两个类之后, 剩下的工作就是向 Solrconfig.xml 文件添加以下内容, 让 Solr “知晓” 新建的函数。

```
<valueSourceParser name="concat"
  class="sia.ch15.ConcatenateFunctionParser" />
```

这里 name 属性值可以自定义, 但是要符合函数查询请求时的函数语法。为了在查询中利用此函数, 可以像表 15.6 的例子那样进行调用。

表 15.6 自定义连接函数的输出样例

函数语法	第一个参数值	第二个参数值	第三个参数值	输出
concat (field1, field2, "-")	"hello"	"world"	"-"	"hello-world"
concat (field1, "Trey", ", ")	"hello"	"Trey"	", "	"hello, Trey"
concat (123, field3, ".")	123	456	."	"123.456"
concat ("no", "delimiter")	"no"	"delimiter"	null	"nodelimiter"
concat (field1, field2, field1)	"hello"	"world"	"field1"	"hellofield1world"

从表中最后一个例子可以看出, 前两个输入使用 ValueSource 类型, 最后一个输入使用字符串类型, 这导致对输入的解析也不同。第一个和最后一个输入内容在语法构成上是一样的, 但是输出的结果不一样。表 15.6 最后一行, 输入 field1 作为第一个参数值和第三个参数值的解读是不一样的。为了提高自定义函数的可重用性 (reusable), 一个不错的想法是在输入参数中尽可能多地使用 ValueSource, 尽量以显性方式传递常量。

本书提供了已配置好的 Solr 内核 customfunction, 你可以用它来测试连接函数。执行以下命令添加一个样本文档：

```
java -Durl=http://localhost:8983/solr/customfunction/update
  -jar post.jar ch15/documents/customfunction.xml
```

当内核 customfunction 可用和样本文档索引好之后 (包含 field1=hello 和 field2=world), 就可以很容易地测试自定义的连接函数了。要在 introduction

文档中创建一个伪字段，其值为 "hello, world!", 提交以下查询请求。

#### 查询请求

```
http://localhost:8983/solr/customfunction/select?q=*&
fl=introduction:concat(concat(field1,field2," "),"!")
```

#### 搜索结果

```
{
  ...
  "response":{ "numFound":1, "start":0, "docs":[
    {
      "introduction":"hello, world!"]}
  ]}
```

至此，我们对 Solr 的函数查询工作原理已经有了较为深入的理解，包括如何通过 Solr 强大的插件功能轻易创建和使用自定义的函数。下一节介绍 Solr 的地理空间搜索功能（仍然会涉及函数的使用），演示如何计算距离和基于坐标系（如地球的经纬度）来搜索形状。

## 15.2 地理空间搜索

基于地理位置的搜索是 Solr 的特色功能之一。实现方式通常是索引每个文档包含地理位置的点（经度和纬度）的字段，然后在查询时请求 Solr 过滤掉没有落在一定半径范围内的点。

Solr 包含两个版本的地理空间搜索功能。旧版本简单一些，支持基于单个经纬度对的半径搜索，返回每篇文档与搜索点的距离，根据距离进行排序。新版本复杂一些，不仅支持基于单个点过滤，还支持基于形状（包括任意复杂多边形）的过滤。旧版本在查询时通过计算文档中点之间的距离来过滤掉距离太远的值。新版本会对索引形状进行索引，形成一系列网格坐标框，在这些索引的网格框中实现对大量文档的搜索，不必再计算距离。旧版本在处理小规模索引和不需要索引外部数据的情况时速度更快，新版本通常在处理大规模文档时速度更快，而且它支持多种形状的搜索，更加灵活。

### 15.2.1 搜索附近的一个点

Solr 的简单地理空间搜索功能可以根据单个点（通常是经纬度）进行搜索，提供简单的语法，支持基于圆的半径或者正方形（比基于圆的计算速度更快，此正方形的边和圆的直径相等）的过滤。

## 定义位置字段

实现半径搜索的第一步是在 `schema.xml` 文件中创建一个字段类型来包含当前地理位置：

```
<fieldType name="location"
  class="solr.LatLonType"
  subFieldSuffix="_coordinate" />
```

`LatLonType` 类用来定义位置字段，它会接受一对经纬度值，并将经度和纬度分别映射到单独的字段。为了将经度和纬度映射到不同的字段，这两个字段需要在 `schema.xml` 文件中也同时存在，可以通过在 `fieldType` 定义中添加 `subFieldSuffix` 结尾的动态字段来实现。本例中的形式是 `"_coordinate"`。

```
<dynamicField name="*_coordinate"
  type="tdouble"
  indexed="true"
  stored="false" />
```

定义好了 `location` 字段，并用一个动态字段代替 `location` 字段将经度和纬度坐标分开映射，下一步就是将文档发送到搜索引擎中，如代码清单 15.4 所示：

### 代码清单 15.4 为地理空间搜索索引位置

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="location">33.748,-84.391</field>
    <field name="city">Atlanta, GA</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="location">40.715,-74.007</field>
    <field name="city">New York, NY</field>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="location">37.775,-122.419</field>
    <field name="city">San Francisco, CA</field>
  </doc>
  <doc>
    <field name="id">4</field>
    <field name="location">37.445,-122.161</field>
    <field name="city">Palo Alto, CA</field>
  </doc>
</add>
```

这些文档可以在随书附带的源代码中找到，执行以下命令就可以轻松将它们发送到 Solr：

```
cd $SOLR_IN_ACTION/example-docs/  
java -Durl=http://localhost:8983/solr/geospatial/update  
-jar post.jar ch15/documents/geospatial.xml
```

一旦 Solr 索引好文档，就可以提供多种位置搜索方式。你可以根据位置的文本名称进行搜索，例如，`city:"San Francisco, CA"`。这种搜索方式的问题在于，无法找到邻近的文档，而待查的地点可能就在城市的边缘。

### 地理位置和边界框过滤器

另一种地理空间搜索方法是查询指定位置一定范围内的所有文档。Solr 包含一个名为 `geofilt` 的查询解析器，接受经纬度坐标、位置字段和最大距离（以千米为单位），匹配位于指定地理空间范围内的文档。搜索距离旧金山市中心 20km 范围内的查询语法如下：

```
http://localhost:8983/solr/geospatial/select?q=*&  
fq={!geofilt sfield=location pt=37.775,-122.419 d=20}
```

`sfield`（空间字段）参数定义了 `schema.xml` 文件中哪个字段包含用于地理查询的 `LatLonType` 字段。`pt`（点）参数定义了经纬度坐标，为半径搜索提供依据。`D`（距离）参数定义了 `geofilt` 过滤器匹配文档的半径距离。图 15.1 演示了当前查询 20km 内的文档区域。

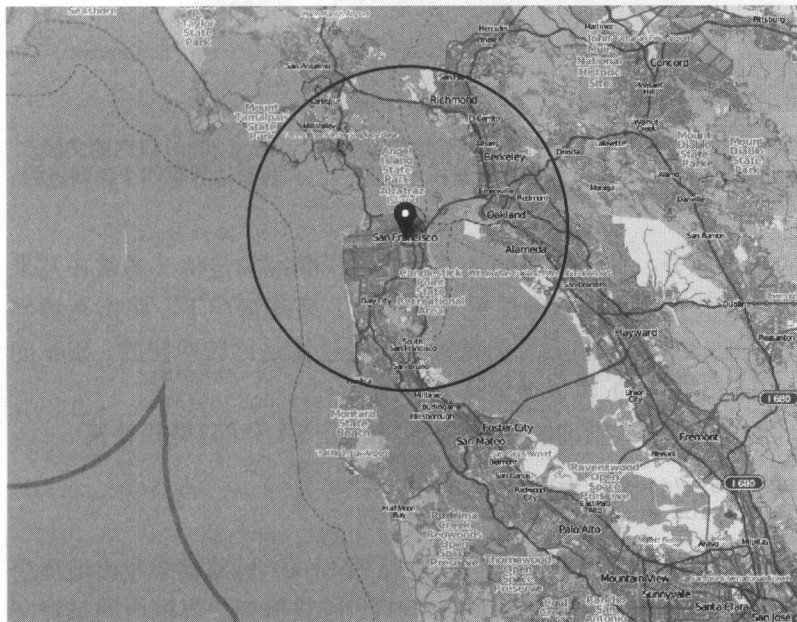


图 15.1 `geofilt` 查询距离美国加州旧金山市中心 20 千米范围内的所有文档（OpenStreetMap.org© 授权提供）

图 15.1 中的 `geofilt` 查询通过两步过滤机制完成。第一步创建一个边界框 (*bounding box*)：正方形的边长等于待搜索范围的直径长度。对文档集使用边界框进行一次过滤，留下来的文档计算出距离，再将边界框以内、圆半径以外的文档过滤掉。图 15.2 演示了以旧金山市为中心，边长为 20 千米的正方形边界框，圆的半径也被计算在了边界框内。

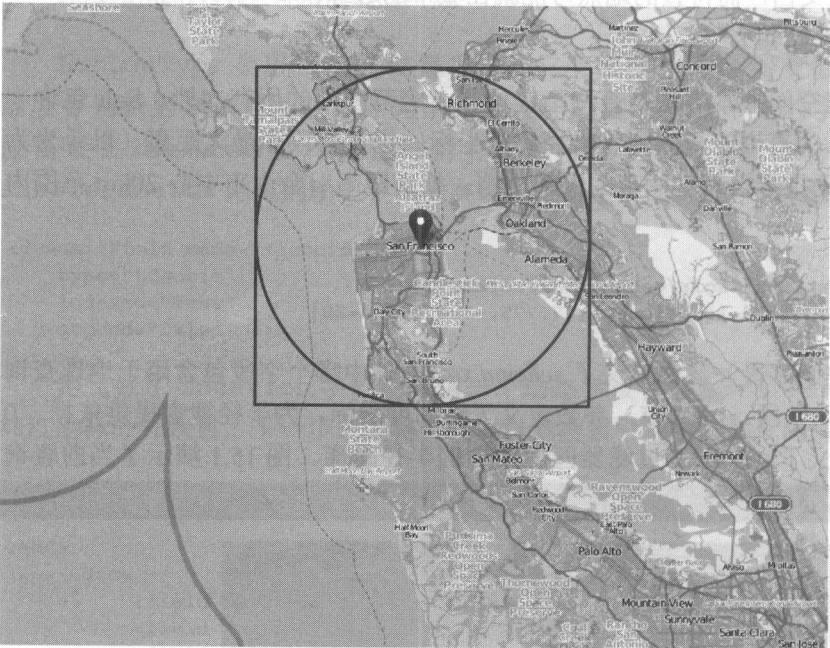


图 15.2 正方形边界框的边长等于圆的直径。边界框内的文档包含圆半径为 20km 以内的文档，当然也可能包含更远距离的文档。边界框过滤之后再进行一次圆过滤，移除边界框四个角附近的文档（© OpenStreetMap.org 授权提供）

`geofilt` 查询的第二部分，关于距离的计算，实现了精准的定位（通常在几米之内）。但是如果文档集很大，计算精确位置非常耗时。某些情况下，将所有边界框内匹配的文档包含进来，放弃花销更大的精确圆半径过滤是比较明智的。`Solr` 的另一个查询解析器 `bbox`（边界框）可以做到这一点。

`bbox` 查询解析器和 `geofilt` 查询解析器的语法相同，两者可以轻松替换：

```
http://localhost:8983/solr/geospatial/select?q=*&fq={!bbox sfIELD=location pt=37.775,-122.419 d=20}
```

至此，我们已经知道如何执行一个快速的边界框 (`bbox`) 过滤器来找出地理空间邻近的位置，也知道如何使用一个更精确的 `geofilt` 过滤器为边界框内的每篇文档计算距离，并根据距离再次过滤半径范围外的文档。针对使用 `geofilt` 过滤器的

查询, Solr 已经为每个匹配文档计算了距离特定点的值, 这一做法方便了在其他地方使用这些距离。特别地, 如果 Solr 可以根据距离或者字段列表中其他字段返回的计算距离对文档进行排序, 那就再好不过了。如你所料, Solr 已经包含以上情况的内置支持。

### 在搜索结果中返回已计算好的距离值

除了基于一个点的距离过滤, Solr 还可以通过 `geodist` 函数在搜索结果中返回每个文档已经计算好的距离。15.1.3 节中提到如何将函数的计算结果返回成伪字段, `geodist` 函数返回的计算结果遵循相同的模型。

`geodist` 函数语法是 `geodist(sfield, latitude, longitude)`。通过这个函数, 可以将任何经纬度与每个文档的距离返回为动态计算的伪字段。代码清单 15.5 演示了如何将每个文档与旧金山坐标点 (37.77493, -122.41942) 之间的距离返回为伪字段。

#### 代码清单 15.5 以为字段形式返回计算的距离

##### 查询请求

```
http://localhost:8983/solr/geospatial/select?q=*&
  fl=id,city,distance:geodist(location,37.77493, -122.41942)
```

geodist 函数  
计算结果赋  
予 distance  
伪字段。

##### 搜索结果

```
{
  ...
  "response": { "numFound": 4, "start": 0, "docs": [
    {
      "id": "1",
      "city": "Atlanta, GA",
      "distance": 3436.669993915123
    },
    {
      "id": "2",
      "city": "New York, NY",
      "distance": 4128.9603389283575
    },
    {
      "id": "3",
      "city": "San Francisco, CA",
      "distance": 0.03772596784117343
    },
    {
      "id": "4",
      "city": "Palo Alto, CA",
      "distance": 43.17493506307893
    }
  ]
}
```

每个文档目前  
包含一个动  
态计算过的  
distance 字段。

结果集中的每个文档都包含一个新的 `distance` 字段。与其他函数查询一样,



我们可以很容易地将返回 `geodist` 函数的计算值作为一个伪字段。这里需要明确一点，两个点之间的空间距离涉及一些复杂的数学计算，因此在百万级文档上的计算速度会变慢。这时候，先使用 `geofilt` 过滤器限制一下结果集通常会好一些。

## 距离排序

15.1.4 节曾提到，不仅可以返回函数计算值，还可以对它们进行排序。在这方面，`geodist` 函数和其他函数没有差别。将 `geodist` 函数添加到 `sort` 参数中，可以实现对文档由近及远的排序，具体操作如代码清单 15.6 所示。

代码清单 15.6 基于地理空间距离的排序

### 查询请求

```
http://localhost:8983/solr/geospatial/select?q=*&
  fl=id,city,distance:geodist(location,37.77493, -122.41942)&
  sort=geodist(location,37.77493, -122.41942) asc, score desc
```

### 搜索结果

```
{
  ...
  "response": { "numFound": 4, "start": 0, "docs": [
    {
      "id": "3",
      "city": "San Francisco, CA",
      "distance": 0.03772596784117343
    },
    {
      "id": "4",
      "city": "Palo Alto, CA",
      "distance": 43.17493506307893
    },
    {
      "id": "1",
      "city": "Atlanta, GA",
      "distance": 3436.669993915123
    },
    {
      "id": "2",
      "city": "New York, NY",
      "distance": 4128.9603389283575
    }
  ]
}
```

← 搜索结果  
首先根据  
`geodist`  
函数排  
序。

← 每个搜索结  
果比上一个  
结果的位置  
距离更远。

该请求会对所有文档按照距离从近到远进行排序，再根据相关度得分由高到低排序（如果有相关度评分，则不同于本例情况）。如果想要先显示最相关的工作，然后再根据距离远近排序，则可以调整排序的先后顺序。通过多个函数查询的组合，也可以将关键词的相关度得分和地理相似度作为不同的因素组合在一起，作为文档的组合相关度得分。第 16 章会介绍一些更高级的基于函数的相关度技术。



### 重用地理参数

目前过滤、返回伪字段和排序的例子都一一演示过了。如果要在一个查询中使用这三种功能，需要使用如下的语句在查询中组合它们：

```
http://localhost:8983/solr/geospatial/select?q=*&
fq={!geofilt sfield=location pt=37.775,-122.419 d=20}&
fl=*,distance:geodist(location, 37.775,-122.419)&
sort=geodist(location, 37.775,-122.419) asc, score desc
```

这样的查询是多余的，它需要用户多次重复输入同样的内容。幸好，`geofilt`、`bbox` 和 `geodist` 函数都可以在查询请求的解引查询字符值中得到它们所需的参数。因此，之前的查询可简化为：

```
http://localhost:8983/solr/geospatial/select?q=*&
fq={!geofilt}&
fl=*,distance:geodist()&
sort=geodist() asc, score desc&
sfield=location&pt=37.775,-122.419&d=20
```

对于任意地理空间组件而言，通过显式声明 `sfield`、`pt` 和 `d` 参数并将其作为本地参数的请求，仍然可以使用简化语法覆盖它们的默认值。对于基本使用而言，由于经常使用相同的点来过滤、排序及返回一个计算距离，因此这个操作可能不是必要的。

## 15.2.2 高级地理空间搜索

除了 15.2.1 节中介绍的简单的单点地理空间搜索功能之外，Solr 还具备更新和更高级的功能。高级功能不仅支持索引一个点，还支持索引包含任意多边形的字段，而且可以索引每个字段中的多个点或多边形。

这些功能为什么如此有用？就支持每个文档中的多个位置来讲，设想一下搜索餐馆的场景，我们知道一些餐馆在全球多个地方有连锁店，此时你不必为每个餐馆的位置建立一个单独的文档，而是可以将多个经纬度坐标索引到一个文档。在查询时，包含查询中一个或多个餐馆位置的文档将会被返回。

形状（圆形、正方形或者其他任意多边形）比单一点更能代表一个文档。例如，如果为政府官员建立文档模型，索引的位置是否就是他们所代表的居民所在的地理区域呢？再比如，如何用文档去表示标志性建筑（如中国的长城）、城市或者国家？你可能会想到用距离一个点一定范围内的区域代表一个地方，不过用多边形来代表它们的地理范围可能更好。使用这种方式搜索 100 千米以内的文档时，会返回 100 千米范围内与查询半径上的点有重合的位置文档，而不是只返回与位置查询中心点重合的文档。

## 基于网格的位置搜索

为了对一个或多个任意形状进行有效索引和搜索，Solr 使 `SpatialRecursivePrefixTreeFieldType` 类调用特殊字段类型，将世界划分为网格。当缩小到一定程度时，世界可划分为由许多部分组成的网格。以四象限为例，每个象限都可以被再次划分为更小的包含四个象限的网格，这种划分可以无限递归下去。图 15.3 演示了三级深度的象限划分。

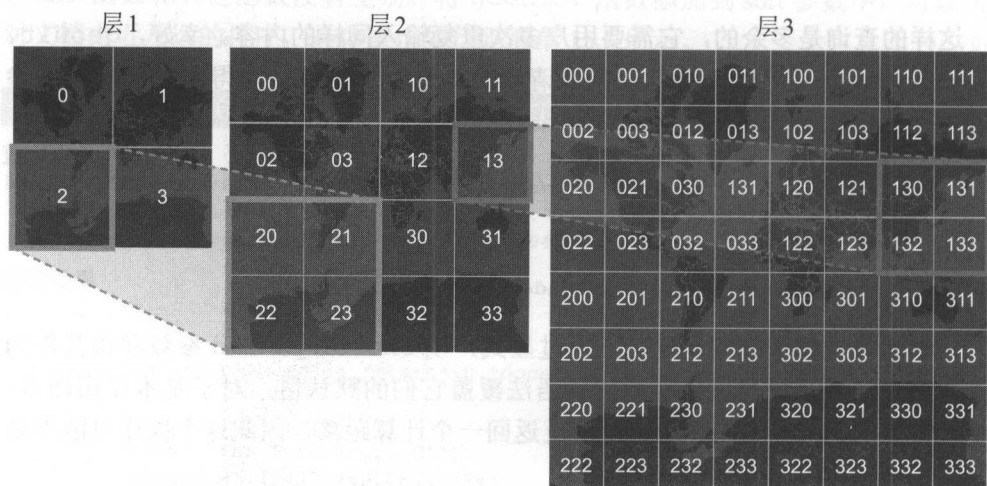


图 15.3 将世界划分为多个层级的四象限。根据最终要达到的精确度需要，每个层级可被再次划分为多个级别的四象限

从图 15.3 可以看出，网格的每个框都包含一个唯一的标识符，每个框的下一层次的网格中又会添加一位标识符来提高精度。这种建模方式对基于索引词的位置搜索更加高效。图 15.3 中佛罗里达州的迈阿密定位在标号为 0、03 和 032 框中，所以这三个层级的框（还可以根据需求达到的精确度级别，添加尽可能多的层级）都会保存在 Solr 的索引中。这将实现强大的查询功能。想要搜索西半球的文档吗？只要搜索 Solr 索引中位置字段 "0" 或 "2" 即可。在索引网格上进行搜索一般会比计算每个文档的距离并过滤文档的方式更快。这是因为后者会将不在半径范围内的文档也计算一遍。想要搜索在澳大利亚的文档吗？只需要对 "310" 或 "311" 框进行查询。很显然，通过图 15.3 的三层网格不能满足任意精度的位置搜索需求，因此需要更多层级（数量可以自定义）的网格来提高位置查询的精确度。图 15.4 给出了一个更具体的例子，演示了如何搜索加州旧金山附近的相关文档。

如图 15.4 所示，如果世界被多层级的网格分割后，查询旧金山区域的“形状”就是找到包含旧金山而不包含其他区域的大小网格的最佳组合。因为每个必需的框都将成为查询的一部分，根据搜索应用精确度要求，最佳组合意味着找到任意

组合层的最小框数。幸好, Solr 对搜索的网格框进行了优化选择, 用户无须考虑这些。虽然用户没必要掌握 Solr 高级搜索功能中网格系统的工作原理, 但还是建议对其有一些基本了解, 这会有助于在索引和查询阶段确定位置精度需求的配置选项, 做出更好的权衡。

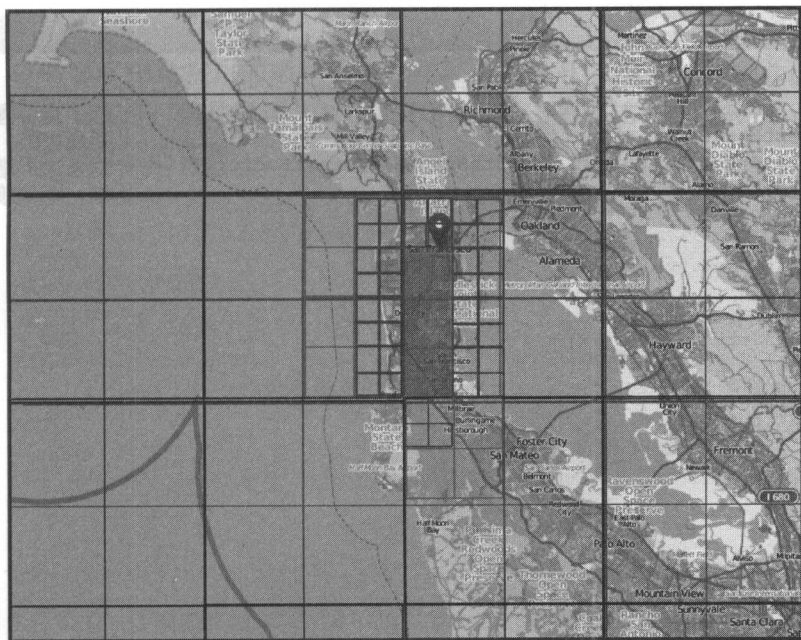


图 15.4 加州旧金山区域构建的网格模型。区域中心部分包含更细粒度的框（更大的框），区域的边缘还需要更多细粒度的网格来接近真实多边形的边（© OpenStreetMap.org 授权提供）

### PrefixTree 方法：GeoHash 和 Quad

基于网格的定位系统使用了前缀树（prefix trees）概念，以一系列或粗略或精确的前缀（例如，31102）来代表地点。SpatialRecursivePrefixTreeFieldType 对前缀树进行抽象表示，允许实现用来索引或搜索空间数据的多层级网格系统的不同功能。以上的例子描述了通过在每个层级上定义 4 个象限的简单的 4 个网格功能。另一个前缀树的功能依存于众所周知的标准 Geohash，将每个层级分为 32 个框而不是 4 个。Geohash 是专门为建立地球空间模型而设计的，在 schema 中定义 SpatialRecursivePrefixTreeFieldType 时，设定 geo=true 启用个功能。Geohash 标准是有据可查的，具体的标准对于理解本章内容是不必要的，它的功能在概念上和 4 个网格功能是类似的。有关 Geohash 的更多内容，请参考 <http://en.wikipedia.org/wiki/Geohash>。

理解基于网格的搜索系统将有助于你理解 Solr 的高级地理空间功能。在索引和查询时将位置映射到网格坐标系的所有复杂操作都会在 `SpatialRecursivePrefixTreeFieldType` 类或它所依赖的类中处理。用户只需在 `schema.xml` 文件中添加一个字段类型，具体操作如代码清单 15.7 所示。

代码清单 15.7 定义一个支持高级地理空间搜索的位置字段

```
<fieldType name="location_rpt"
  class="solr.SpatialRecursivePrefixTreeFieldType"
  spatialContextFactory=
    "com.spatial4j.core.context.jts.JtsSpatialContextFactory"
  distErrPct="0.025"
  maxDistErr="0.000009"
  autoIndex="true"
  units="degrees" />
<field name="location_rpt"
  type="location_rpt"
  indexed="true"
  stored="true"
  multiValued="true" />
```

最大可接受距离误差，这决定了需要计算的网格级别数量。

形状的精确度介于 0.0（完全精确）与 0.5（不精确）之间。

除了点、圆、矩形之外，对多边形增加一个依赖项。

允许每个字段支持多个点 / 形状。

定义好字段后，就可以将示例形状和点索引到这个字段中。

### 点

一个点可以通过传统的逗号分隔来定义，如：`latitude,longitude`，也可以通过空格来定义，如 `latitude longitude`。以下两个例子的结果是一样的。

```
<field name="location_rpt">43.17614,-90.57341</field>
<field name="location_rpt">-90.57341 43.17614</field>
```

### 矩形

矩形通过用代表角的四个点来索引。这些点可以用 `MinX`，`MinY`，`MaxX`，`MaxY` 顺序代表：

```
<field name="location_rpt">-74.093 41.042 -69.347 44.558</field>
```

### 圆形

一个圆通过圆心和半径来定义。为了明确这是一种更复杂的形状，输入参数被包装在了一个简单的语法以定义该形状。

```
<field name="location_rpt">Circle(37.775,-122.419 d=20)</field>
```



圆的圆心使用点的语法来定义，输入形式可以是 `latitude,longitude`，也可以是 `latitude longitude`。圆的半径通过 `d` 参数来定义，代表距离（不要和直径混淆）。

### 其他形状

为了支持任意复杂的形状，Solr 支持通过 WKT (well-known text) 标准定义形状。虽然 Solr 中最常用的形状是点、圆形、矩形可以通过 Apache 2.0 Licensed Spatial4J 库原生支持，但是通过 WKT 定义多边形需要在 JTS (the Java Topology Suite) 中添加一些可选的依赖。

### JTS 的开源许可和合法用途

Apache Solr 是开源的，采用 Apache 2.0 License 许可，用户有权在任何系统（包括私有系统）中使用 Solr 的代码和软件，分享代码没有任何法律责任，无须支付许可费。

Solr 通过不那么宽松的 LGPL（一种较不通用的公共许可）授权允许集成 JTS。JTS 库（JAR 文件）并不随着 Spatial4J 包含在 Solr 中，这使得 JTS 作为一个可选的依赖不需要使用 Solr 中基本的形状（点、矩形和圆）。使用 Solr 中的 JTS 需要用户自行添加依赖包。

只要不改变 JTS 中的代码，引用 JTS 类库，不申请 LGPL 许可（要求派生作品开放应用程序的源代码）是安全的。引用 LGPL 中任何一个类都需要咨询公司政策或法律顾问，以确保合法使用。如果只需要 Solr 支持点、圆形和正方形，在定义地理字段类型的时候就不需要使用 JTS 的 `spatialContextFactory` 功能。

为了实现本节描述的 WKT 功能，需要将 JTS JAR 添加到 `solr.war` 文件中，这样当 Solr 第一次启动时，`servlet` 容器就可以加载 JTS 依赖。因为 JTS 并不和 Solr 关联（由于许可问题），因此需要向 `WEB-INF/lib/` 文件夹中添加 JTS JAR 文件（“`jts.jar`”）。命令如下：

```
cd $SOLR_INSTALL/example/webapps/  
cp -r $SOLR_IN_ACTION/example-docs/ch15/jts/ ./  
jar -uf solr.war WEB-INF/lib/jts.jar  
rm -rf WEB-INF/
```

一旦更新 `solr.war` 文件来包含 JTS 依赖，就可以在 `schema.xml` 文件中引用 JTS 库，如代码清单 15.7 所示。关闭 Solr（如果它在运行状态），执行以下命令来更新 schema 并重启 Solr。

```
cd $SOLR_INSTALL/example/
cp solr/geospatial/conf/jts_schema.xml solr/geospatial/conf/schema.xml
java -jar start.jar
```

Solr开启并在JTS的支持下运行时，可以演示一些强大的索引搜索形状的地理空间功能。

WKT支持通过以下格式来定义任意多边形。

```
<field name="location_rpt">
  POLYGON((-10 30, -40 40, -10 -20, 40 20, 0 0, -10 30))
</field>
```

使用这个语法定义任意多边形的时候需要注意以下几项：

- 点与点之间是通过逗号分隔的。
- 每个点的格式是 longitude latitude。
- 第一个点和最后一个点的定义是一样的，表示多边形的首尾闭合。
- WKT 点集被影射到有效的经度 ( $\pm 90^\circ$ ) 和纬度 ( $\pm 180^\circ$ ) 范围内。
- 支持国际日期变更线，但仅圆形支持极点封装。

通过索引任意多边形，Solr 为复杂的基于位置的搜索提供了支持。基于多边形的搜索是 Solr 空间搜索功能的强大之处。

## 多边形查询

15.2.1 节介绍过如何使用 LatLonType 字段和的基于单个点的地理空间功能来实现半径和边界框搜索。通过使用 15.2.1 节中讨论的 geofilt 和 bbox 查询解析器在 SpatialRecursivePrefixTreeFieldType 字段中也可以实现类似搜索：

```
http://localhost:8983/solr/geospatial/select?q=*&
fq={!geofilt pt=37.775,-122.419 sfield=location_rpt d=5}

http://localhost:8983/solr/geospatial/select?q=*&
fq={!bbox pt=37.775,-122.419 sfield=location_rpt d=5}
```

除了描绘圆和边界框，还可以查询使用 SpatialRecursivePrefixTreeFieldType 提供的基于网格前缀系统产生的可被索引的形状。这类查询的语法如下：

```
http://localhost:8983/solr/geospatial/select?q=*&
fq=location_rpt:"Intersects(-90 -90 90 90)"
```

该查询搜索了一个矩形，将会返回 location\_rpt 字段中与查询的矩形有交叉的所有文档。

除了 Intersects 操作，SpatialRecursivePrefixTreeFieldType 字段中的查询还支持 IsWithin、Contains 和 IsDisjointTo 操作，表 15.7 中定义了这些操作。

表 15.7 高级形状查询和形状操作

形状操作	说明
Intersects	与查询中的形状有所交叠的一个或多个形状的匹配文档
IsWithin	文档中的一个或多个形状包含在查询中的形状里
Contains	文档中的一个或多个形状完全包含查询中的形状
IsDisjointTo	返回不包含查询中的形状的文档

通过表 15.7 中的操作，可以很快找到完全包含在一个特定多边形中的所有形状。

```
http://localhost:8983/solr/geospatial/select?q=*:*&
  fl=id,location_rpt,city&
  fq=location_rpt:"IsWithin(
    POLYGON((
      -85.4997 34.7442,
      -84.9723 30.6134,
      -81.2809 30.5255,
      -80.9294 32.0196,
      -83.3024 34.8321,
      -85.4997 34.7442))
  ) distErrPct=0"
```

这个查询是搜索约为美国佐治亚州（GA）大小的多边形中包含的所有文档。图 15.5 直观地演示了通过定义的多边形来搜索的区域。

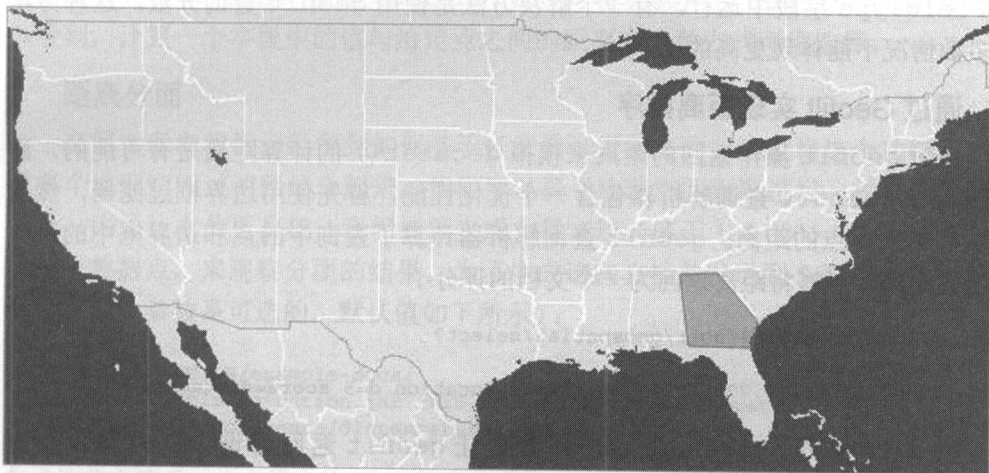


图 15.5 加利福尼亚州形状下的多边形区域搜索

运行查询后会发现只有一个文档落在了亚特兰大：



```
{
  ...
  "response":{"numFound":1,"start":0,"docs":[
    {
      "id":"1",
      "location_rpt":["33.748,-84.391"],
      "city":"Atlanta, GA"]}
  ]}
```

可以根据空间数据的情况通过标准的布尔逻辑运算来构建复杂的查询操作。我们不仅可以找到匹配查询的文档，还可以通过索引形状和查询形状之间的距离来影响结果的相关度。第 16 章将会深入介绍使用空间距离影响查询（q 参数）的相关度评分，现在有必要看一下怎样返回来自 geofilt 操作的值。

### 利用 PatialRecursivePrefixTreeFieldType 基于距离排序

对于 Solr 4.5, geodist() 函数可以在 SpatialRecursivePrefixTreeFieldType (geodist() 除了支持 LatLonType, 也支持它) 中运行。然而, 当使用前缀树功能的时候, 形状的索引缺失了一些精确度会影响距离计算。解决这个问题, 可以将 distErrPct 的值设置得很高, 但是仍存在问题, 即当搜索一个被索引的确切点时, 从 Solr 中的出距离为 5.8165506E-6 样子的数而不是 0, 这会有点奇怪。在 Solr 4.5 之前的版本中, geodist() 函数不能在 SpatialRecursivePrefixTreeFieldType 字段中运行, 有一个解决方法是使用 geofilt 查询分数, 这种方法在多数情况下能体现更高的性能。

### 通过 Geofilt 实现距离排序

通过 geofilt 操作返回的距离来模拟 geodist() 的计算距离是有可能的。回顾图 15.2, geofilt 查询解析器包含一个优化性能, 首先使用边界框过滤器, 然后计算框中所有点的距离。geofilt 查询解析器计算了查询中的点和边界框中的所有文档的距离, 可以将距离返回为一个文档的评分:

```
http://localhost:8983/solr/geospatial/select?
  sort=score asc&
  q={!geofilt pt=37.775,-122.419 sfield=location d=5 score=distance}
```

geofilt 中的 score=distance 参数让 geofilt 返回查询中的点与每个文档在 geo 字段中的点之间的距离。geofilt 是主查询 (q 参数) 的一部分, 文档的评分等于每个文档计算出的距离。除了 score=distance, 还可以提交请求 score=recipdistance, 这将会返回距离的倒数, 因而最近的文档排名最靠前。如果没有提交 score 参数, 距离也就没必要计算 (对于 SpatialRecursivePrefixTreeFieldType 字段来讲), 每个文档都会从 geofilt 请求中得到相同的常量分值。

如果想要为所有的文档(包括落在 geofilt 计算结果之外的那些文档)计算分值,也可以关闭查询解析器 geofilt 的过滤功能:

```
http://localhost:8983/solr/geospatial/select?
  sort=score asc&
  q={!geofilt pt=37.775,-122.419 sfield=location_rpt d=5
    score=distance filter=false}
```

通过关闭 geofilt 中的过滤功能(听起来虽奇怪不过确实有用),可以让 geofilt 像 geodist 函数一样起作用,这时它会返回计算的距离并不应用过滤器。但是,由于 geofilt 不是一个函数查询,所以对于独立于主查询的分值来排序的函数,如果想要使用它,则需要在一个 query 函数中包含 geofilt 查询,具体操作如下:

```
http://localhost:8983/solr/geospatial/select?
  sort=$distance asc&
  fl=id,distance:$distance&
  q=:*:*&
  distance=query($distFilter)&
  distFilter={!geofilt pt=37.775,-122.419 sfield=location_rpt d=5
    score=distance filter=true}
```

请求将 geofilt 传递给 query 函数,最终返回 distFilter 操作(此例中,离搜索点最近的距离在 geofilt 中指定)的值。这个被间接引用的参数随后可以传递给 sort 和 fl 参数,使用 LatLongType 时将会被用作 geodist() 函数。可以看到,计算一个字段中的值与给定点之间的距离有多种方式可供选择。

## 距离分面

在第 8 章中我们已经学到如何基于任意查询进行分面。通过将这个功能和一个或多个地理空间过滤器结合起来,我们可以基于地理空间距离进行一些有趣的数据分析。为了让本节更有趣,我们打算生成大量文档(多于 100 000),而不是基于一小部分数据点,来观察分面的结果。为了生成这些文档并发送到 Solr 中,执行以下命令(file 参数是可选的,默认值如下所示):

```
cd $SOLR_IN_ACTION/example-docs/
java -jar ../solr-in-action.jar ch15.DistanceFacetDocGenerator
  ➡ -file ch15/documents/distancefacet.xml
```

生成这些文档后(将占据大约 15MB 的磁盘空间),可以通过执行相同目录中的如下命令将文档发送到 Solr:

```
java -Durl=http://localhost:8983/solr/distancefacet/update
  ➡ -jar post.jar ch15/documents/distancefacet.xml
```

你将在控制台窗口看到创建的文档。每个文档都包含一个地点的纬度和经度,以及城市的名称。有了这些信息可以创建分面,展示文档集与任意给定点的全部距

离。例如，为 10km 以内、20km 以内、50km 以内、100km 以内的所有文档生成分面查询。

这里介绍一个更复杂的例子。假如要运行一个分面，得到在 50km 半径范围内的前 10 个城市，之后运行第二个查询得到 20km 半径范围内的前 10 个城市听起来有些复杂？代码清单 15.8 给出了查询过程。

### 代码清单 15.8 运行一个半径搜索返回匹配查询的前 10 个城市

#### 查询请求 1

```
http://localhost:8983/solr/distancefacet/select?q=*&rows=0&
fq={!geofilt sfield=location pt=37.777,-122.420 d=80}&
facet=true&
facet.field=city&
facet.limit=10
```

① 在大半径里只取得排名靠前的 10 个城市……

#### 搜索结果 1

```
facet_fields:{
  "city":[
    "San Francisco, CA",11713,
    "San Jose, CA",3071,
    "Oakland, CA",1482,
    "Palo Alto, CA",1318,
    "Santa Clara, CA",1212,
    "Mountain View, ca",1045,
    "Sunnyvale, CA",1004,
    "Fremont, CA",726,
    "Redwood City, CA",633,
    "Berkeley, CA",599]
```

第 2 个城市：San Jose, CA。

第 1 个城市：San Francisco, CA。

第 10 个城市：Berkeley, CA。

#### 查询请求 2

```
http://localhost:8983/solr/distancefacet/select?q=*&rows=0&
facet=true&
fq=(
  _query_:"{!geofilt sfield=location
    pt=37.777,-122.420 d=20}"

  OR _query_:"{!geofilt sfield=location
    pt=37.338,-121.886 d=20}"

  ...
  OR _query_:"{!geofilt sfield=location
    pt=37.870,-122.271 d=20}"
)&
facet.query={!geofilt key="san francisco, ca"
  sfield=location
  pt=37.7770,-122.4200 d=20}&
facet.query={!geofilt key="san jose, ca"
  sfield=location
  pt=37.338,-121.886 d=20}&
...
facet.query={!geofilt key="berkeley, ca"
  sfield=location pt=37.870,-122.271 d=20}
```

第 2 个城市：San Jose, CA。

② 在小半径内取得排名靠前的 10 个城市……

第 1 个城市：San Francisco, CA。

第 10 个城市：Berkeley, CA。

③ 得到排名靠前的 10 个城市的文档的分面统计数。

第 1 个城市：San Francisco, CA。

第 2 个城市：Berkeley, CA。

初始查询是找到距旧金山中心纬度/经度坐标点 50km 之内的前 10 个城市 ❶ (按数量排序)。基于这些结果,可以在地图上绘制这些城市的经纬度坐标,然后画出距每个城市的中心 20km 范围内的圆。运行第二个查询后,可以知道距每个城市 ❷ (即使它们属于第一个查询以外) 20km 之内的所有文件的数量,这样我们就可以将这些结果展示给用户,也可以用深色阴影覆盖 20km 以内有更多数量文档的城市 ❸。

在图 15.6 中可以看出, 由于是执行两次搜索后得到的结果, 第二次搜索得到的位置超出了第一个查询半径的制约。第一个查询可以用来发现候选城市, 这些城市可以产生第二个分面查询。众多像这样产生分面组合的方法可以提高应用程序的复杂地理空间分析功能。

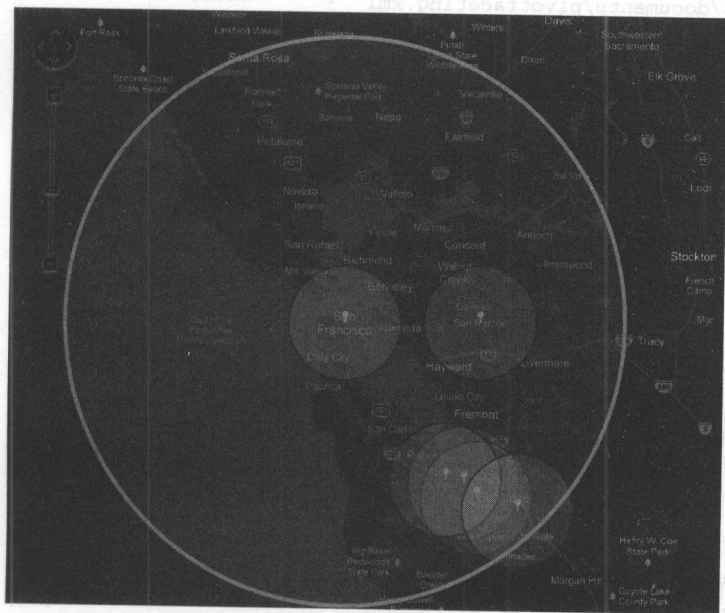


图 15.6 基于地理空间距离的分面可视化

至此，Solr 支持一些奇妙的地理空间用户体验的查询工具已经全部介绍完毕。下一节将跳出空间搜索功能，聚焦 Solr 各种有趣的数据分析能力。

### 15.3 分面透视

第 8 章已经介绍了所有关于 Solr 分面的知识，以及如何基于它展开有趣的数据分析和数据可视化。所有在第 8 章演示的分面例子（字段、排序以及查询分面）返回的是基于单个字段或查询值的总数量。然而，在很多情况下需要嵌套分面：能够基于多个字段值（功能类似于现代电子表格应用的数据透视表）返回聚合总数量的

分面。Solr 支持一种高级的分面形式——分面透视（或称为决策树分面），这种分面可以提供基于多个字段值的条件分面计数。

例如，Solr 索引包含 restaurant 文档的三个字段：state、city 和 rating。rating 字段代表 1~5 星的评分。如果想看看前三个州的前三个城市中有多少 4~5 星的餐厅，可以如代码清单 15.9 所示通过分面透视来实现。

### 代码清单 15.9 三个字段的分面透视

#### 向 Solr 发送文档

```
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/pivotfaceting/update
    -jar post.jar ch15/documents/pivotfaceting.xml
```

#### 查询请求

```
http://localhost:8983/solr/pivotfaceting/select?q=*&
fq=rating:[4 TO 5]&
facet=true&
facet.limit=3&
facet.pivot.mincount=1&
facet.pivot=state,city,rating
```

① 在 3 个字段上请求一个透视分面。

#### 搜索结果

```
{
...
  "facet_counts":{
    ...
    "facet_pivot":{
      "state,city,rating":[{
        "field":"state",
        "value":"GA",
        "count":4,
        "pivot":[{
          "field":"city",
          "value":"Atlanta",
          "count":4,
          "pivot":[{
            "field":"rating",
            "value":4,
            "count":2},
            {
              "field":"rating",
              "value":5,
              "count":2}]]},
        {
          "field":"state",
          "value":"IL",
          "count":3,
          "pivot":[{
            "field":"city",
            "value":"Chicago",
```

③ 进一步分面细分时统计数会减少。

② 透视分面显示了州、市与评分的层级统计数。

```

        "count":3,
        "pivot":[{
            "field":"rating",
            "value":4,
            "count":2},
            {
            "field":"rating",
            "value":5,
            "count":1}]]},
    {
        "field":"state",
        "value":"NY",
        "count":3,
        "pivot":[{
            "field":"city",
            "value":"New York City",
            "count":3,
            "pivot":[{
                "field":"rating",
                "value":5,
                "count":2},
                {
                    "field":"rating",
                    "value":4,
                    "count":1}]]}]
    ]}]

```

该代码清单展示了一个三级分面透视，由 facet.pivot 参数定义 ❶，其中包含一个由逗号分隔的三个字段列表。不同于第 8 章中介绍的传统分面搜索，传统的仅仅显示单一字段值的计数，分面透视通过另一字段的值来再分割每个分面的值，然后再被另一字段细分，等等。这使得在一个单独的查询中执行分析功能，而不需要遍历第一个查询中的所有值并用另一个字段来执行单独查询来细分分面。此例中，数据由 state、city 及一个 rating 字段实现分面 ❷。随着分面被更具体的值进一步细分，分面的个数会受到限制 ❸。相反，在社交网络配置文件的索引中，用户可能会通过性别、学校、学历或者甚至公司职位等来划分文档。通过将信息按不同的维度分面透视，可以发现隐藏在文件间关系中的丰富知识。

### 分面透视的局限性

分面透视在实用性功能上也有它的局限性。由于一个分面透视请求是根据另一个分面来细分原始分面，计算每一个增加的分面额外的性能（速度）代价就像在单独请求各分面一样。我们一般比较关注在分面透视结果中的返回大量数据的代价。例如，代码清单 15.9 中请求前 3 个州中前 3 个城市的收视率前 3 名的内容，这将会返回 27 个独立的值（ $3 \times 3 \times 3$ ）。如果仅仅将各个级别方面的数量要求由 3 提高 7，就会返回 1000 个不同的分面值。

分面透视的响应格式比传统的分面更为详细，返回的响应大小为若干 MB。如



果不确定每一层返回多少值，那些必须对响应结果进行读取与解析，可能会严重破坏用户的 Java 堆 (heap)，降低查询速度，甚至可能会拖垮应用程序堆栈。鉴于这些原因，需要在设置每个字段的返回值和分面透视的字段数量时多加考虑。

Solr 分面透视的另一个局限是，它目前只能在单个 Solr 内核上进行字段分面。分面透视也不支持区间和查询分面。分面透视的这两个局限性更多体现出分面透视功能还不够成熟，而不是技术方面存在挑战。分布式分面透视有一个补丁 (<https://issues.apache.org/jira/browse/SOLR-2894>)，可能不久后会成为 Solr 新版本的一部分，Solr 未来版本也会扩展可供分面透视使用的分面类型。

### 分面透视的未来改进

Solr 一直是受欢迎的实时数据分析工具，分面会随着更多维度的增加而成熟。例如，正在执行的几个统计工作，包括每一个数据透视级别字段的求和、求平均数及求百分位上的数值 (<https://issues.apache.org/jira/browse/SOLR-3583>)；还有正在积极开发、支持对多种搜索结果进行统计分析的“分析组件” (<https://issues.apache.org/jira/browse/SOLR-5302>)。此外，有一些关于分面透视是否应该从传统分面独立出来，或者是否所有分面都应该是分面透视的讨论。毕竟，除了响应格式的差异，传统分面仅被认为是单一维度的分面透视。

在不同层级下对多种类型的分面进行透视，将会带来极大的灵活性。首先在 state 字段上分面，然后再深入对 10、25 和 50+ 千米上的分面查询进行限定，接着再通过一个最终数字或日期字段的范围分面来进一步细分这些值，这种情况是可能的。虽然这种分面类型尚不可用，但是它已经获得一些合理的支持，且是未来分面透视发展的大方向。然而目前仅限于字段分面的处理工作，除非能找到并应用一个补丁来尝试一些未来的功能。目前的分面透视有着强大的数据分析能力，它可以仅通过向 Solr 提交一个简单的请求来完成许多嵌套或有条件总数的请求。

Solr 不仅仅提供了嵌套的分面透视，同时也提供了在查询中嵌套查询（将一个查询的结果作为另一个查询的查询或过滤条件）和引用外部数据的功能，后续章节会提到它们。

## 15.4 引用外部数据

查询时在 Solr 索引中主要有三种机制可以引用外部数据。第一种是执行自定义函数中的代码找到外部数据源。虽然本书中没有专门演示这个过程，不过只需要知道如何编写自定义函数（参见 15.1.6 节），就可以轻松地将代码直接插入到引用 Solr 索引之外数据的自定义函数中。

如果引用数据是为了确定文档之间的关系，那么就可以使用 Solr 内核之外的数



据的第二种方法，即与同一台服务器上的另一个 Solr 内核进行跨文档连接。例如，如果将结果集限制为只让当天的特定用户看到的文档，则可以保持一个独立的 Solr 内核来建立用户和文档之间的映射（而不必修改每个用户所查看信息的源文件）。通过这种设置可以建立 Solr 内核之间的连接，从而将主查询中返回的内容限制为在第二个 Solr 内核的子查询中未找到的文档。15.5 节会简要介绍如何执行这类跨文档和跨内核的连接。

第三种机制是在查询中使用特殊字段类型——ExternalFileField 来使用 Solr 索引之外的数据，ExternalFileField 可以从外部文件加载数值。

### 使用 Solr 的 ExternalFileField

ExternalFileField 字段允许用户独立于 Solr 索引过程来更新文档中的特定字段。这是通过在 Solr 服务器中指定文本文件来完成的，这个文本文件包含特定外部字段值和每个文档 ID 的映射。如果想在单个字段中批量更新所有文档而又不重新索引文档，这个方法将是很有用的。

定义一个外部文件字段就像定义 schema.xml 文件中的其他字段一样，但是会有一些额外的参数：

```
<fieldType name="popularity" keyField="id" defVal="0"
  stored="false" indexed="false" class="solr.ExternalFileField"
  valType="pfloat"/>
```

上面定义的 popularity 字段从每个外部文件的文档中取值（不是从 Solr 索引和其他字段中取值），因此必须把文件放在 Solr 服务器上。外部文件应位于 Solr 内核中定义的 Solr 索引目录下（默认是 data/）。文件名字格式必须是 external\_{field} 或者 external\_{field}.\*，其中 {field} 是 ExternalFileField 在架构中定义的名称。对于已经定义的字段，这些文件可以被命名为 external\_popularity、external\_popularity.txt 或者 external\_popularity.2013\_11\_03\_09\_00。如果一个字段中存在多个文件，则将会使用字典排序中的最后一个。这样在更新文件时是很方便的，因为用户可能需要保留旧版本或者无法覆盖仍然在使用的旧文件。

external\_popularity 文件的示例如下：

```
doc123=22
doc456=15.7
doc789=19.4
```

在这个例子中，doc123 中的 popularity 字段从外部文件取值时都会返回 22。ExternalFileField 有几个属性值得注意。

第一，该字段的值不能被直接搜索；它们仅限于在返回字段列表（fl 参数）或查询函数中使用。虽然技术上可以实现通过 frange 过滤器（见 15.1.2 节）来筛选

一个值或一个范围的值，但这个会比直接索引查找慢。

第二，不是外部文件的所有值都需要映射到索引值，也不是所有索引值都需要映射到外部文件的值。如果没有与文件的键值相匹配的，则该文件中的 `ExternalFileField` 的值将是在 `schema.xml` 中该字段的默认值。

第三，`ExternalFileField` 字段目前仅支持浮点型数值（为 `valType` 使用 Solr 的浮点类型），不支持其他数据类型。因此，这个严重限制了 `ExternalFileField` 的用途。在实际用途中，`ExternalFileField` 值能够很容易地被替换为从外部文件中取值的自定义函数。如果需要 `ExternalFileField` 支持不是浮点类型的外部值，请阅读 15.1.6 节找出最优的解决办法。

第四，当首次从外部文件中读取字段时，`ExternalFileField` 字段值会被写入内存，同时该值不会因为文件更新或命令提交而自动重载。为防止在利用 `ExternalFileField` 字段进行首次查询时太慢，或者希望每次提交命令时文件会重新加载，可以分别给 `first-Searcher` 或者 `newSearcher` 事件添加一个事件监听。为了实现这个功能，需要在 `Solrconfig.xml` 文件中添加以下内容：

```
<listener event="newSearcher"
  class="org.apache.solr.schema.ExternalFileFieldReloader"/>
<listener event="firstSearcher"
  class="org.apache.solr.schema.ExternalFileFieldReloader"/>
```

`ExternalFileField` 字段（和可选的自定义查询函数）不仅可以有效地将外部数据源添加到文档中，还可以将一个查询的结果限制在特定的文档中，即通过在另一个字段或文档集中运行子查询所得到的文档。下一节将介绍如何运用伪连接查询来执行文档间或独立的 Solr 内核之间的子查询。

## 15.5 跨文档和跨索引的连接

3.4.1 节中指出 Solr 文件是非相关的，每个用户文档中的内容都是非规范化的。非规范化意味着每个文档都必须包含查询需要的所有信息的字段，即使该信息可能会在多个文档中被复制。

虽然这种非规范化的文档的方式是设计搜索索引的最佳实践，但是有时候这么做却是不切实际的。对于这些特殊情况，Solr 提供了一个基本的连接功能，这个功能能够找到包含与单独的查询结果字段的值相匹配的字段的文档。

### 跨文档连接

Solr 的连接相当于一个 SQL 嵌套查询，其中主查询的结果来自独立的子查询。Solr 连接查询的语法是：

```
/select?
  fl=RETURN_FIELD_1, RETURN_FIELD_2&
  q={!join from=FROM_FIELD
    to=TO_FIELD}CONSTRAINT_FIELD:CONSTRAINT_VALUE
```

作为对比，与上述语法等效的 SQL 语句为：

```
Select RETURN_FIELD_1, RETURN_FIELD_2 FROM join-data
WHERE TO_FIELD IN (
  SELECT FROM_FIELD from join-data
  WHERE CONSTRAINT_FIELD = 'CONSTRAINT_VALUE'
)
```

子查询得到的数据无法返回到主查询的结果中，这种 Solr 连接查询不是传统 SQL 意义上的连接。相反，只会返回从主查询的结果得到的文档，子查询只用于限制主查询的结果。然而，Solr 查询的这种约束对于不同文档的子查询的结果是非常有用的。

在什么情况下使用 Solr 的连接功能呢？假设有基本的文档，比如餐厅，并需要跟踪特定用户对哪些餐厅采取了行动（点击、评论或访问），那么每时每刻根据用户的操作来更新每一个餐厅文档显然是很痛苦的。除非创建了单独的用户行为文档模型，不然跟踪各种各样的个人和餐厅的行为模式也是很有挑战的。使用 Solr 的连接功能，可以创建独立的文档集来模拟这种行为：一个用于用户的操作，一个用于餐厅的行为。

在最后的连接查询例子中，读者可能会注意到，两个文档集在同一个 Solr 内核（被称为 join-data）中。由于运行一个子查询需要处理大量的数据，Solr 要求连接查询中的所有文档都在同一个服务器上。但这并不意味着必须把所有的文档都塞进同一个 Solr 内核。只要包含连接数据的 Solr 内核在同一个 Solr 实例中，就可以进行跨内核连接。

### 跨内核连接

回到餐厅文档和与之相关的用户行为文档的例子，你可能想将餐厅文档和用户行为文档分在单独的 Solr 内核中，从而使单独查询或单独更新更容易，不会影响另一个文档集的索引。Solr 的连接查询解析器支持更多的 fromIndex 和 toIndex 的参数来提供跨内核连接。假设餐厅和用户行为文档的 schema 如下所示：

#### 餐厅内核的 schema.xml

```
<field name="id" indexed="true" stored="true" />
<field name="restaurantname" indexed="true" stored="true" />
<field name="description" indexed="true" stored="false" />
```

## 用户行为内核的 schema.xml

```
<field name="id" type="string" indexed="true" stored="true" />
<field name="userid" type="string" indexed="true" stored="true" />
<field name="restaurantid" type="string" indexed="true" stored="true" />
<field name="actiontype" type="string" indexed="true" stored="false" />
<field name="actiondate" indexed="true" stored="false" />
```

为了测试一个跨内核连接，我们在 join\_restaurants 内核和 join\_useractions 内核中为示例文档建立索引：

```
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/join_restaurants/update
➡ -jar post.jar ch15/documents/join_restaurants.xml
```

```
java -Durl=http://localhost:8983/solr/join_useractions/update
➡ -jar post.jar ch15/documents/join_useractions.xml
```

给示例文档建好索引后，现在可以建立两个内核之间的连接了。通过一个跨内核连接来返回特定用户在过去两周内评价过的所有印度餐馆名称，查询如下所示：

```
http://localhost:8983/solr/join_restaurants/select?
  fl=restaurantname,text&
  q="Indian"&
  fq={!join fromIndex=join_useractions
      toIndex=join_restaurants
      from=restaurantid
      to=id}userid:user123 AND actiontype:clicked
      AND actiondate:[NOW-14DAYS TO *]
```

这个查询在 join\_useractions 内核中运行一个关于行为标准（过去 14 天中 user123 的点击行为）的子查询，然后限制 join\_restaurants 内核上的文档只能是默认字段为 Indian 的文档，同时要匹配在 join\_useractions 索引子查询中的 restaurantid。如你所见，对于两个通过外键联系在一起的文档，而且它们能分别对另一个文档集发出子查询来限制对方，来保留这两个单独的文档集是可行的。因为连接子查询是一个查询解析器，所以可以通过多个过滤器（fq 参数）在一个请求中调用多个连接，每个都包含一个单独的连接操作。另外，如果在每个连接查询中使用嵌套查询（参见第 7 章），可以通过任意复杂的布尔逻辑来组合多个连接查询，每一个嵌套查询的查询解析器都是独立调用的，好像每一个查询单元都被认为是主查询那样。

虽然不能从正在连接的文档集中取值是一个局限，但是基于单独查询中的其他文档的值 Solr 的连接功能在限制查询结果时是 very 有效的。应该尽量将文档非规范化，而不是频繁地依靠 3.4 节中讨论过的 Solr 连接功能，但连接功能肯定能改善更新多

个文档之间的非标准化值时带来的不合理计算问题。连接功能是 Solr 复杂查询操作的工具箱中的一个。

除了连接查询解析器之外，也有一些支持连接操作的高级查询解析器，如块连接子查询解析器（`{!childof=...}`）和块连接父查询解析器（`{!parentof=...}`）。这些解析器可以更快地连接数据，使用时只需在索引时定义文件之间的父子关系，以便它们能够被添加到搜索索引中。这里我们不会使用这些查询解析器覆盖语义，但是如果标准的连接查询解析器对你来说性能太差，可以选择复杂又快速的替代品——块连接。

## 15.6 使用Solr做大数据分析

虽然 Solr 开始只是作为匹配和排序文档的全文搜索引擎，但是近几年的发展让 Solr 有更大的用武之地。企业现在使用 Solr 作为 NoSQL 数据库、缓存层或分类引擎，甚至作为推荐引擎。Solr 日益增长的使用领域之一就是大数据分析领域。许多组织使用 Hadoop 做海量数据分析。这种方式适用于离线处理，而不允许用户钻入实时数据，所以在预处理领域未占有一席之地。Hadoop 一般在实时场景中（例如在一个网站上）运行速度不够快，不足以保持用户的注意力，因此它不能作为一个好的对用户的特殊查询结果进行实时分析的传送装置。

用户可以使用 Solr 来搜索任意关键词或其他有价值的任何索引字段，也可以看到在几十或几百毫秒之内匹配的数百万甚至数十亿的文档查询总数量。正如第 8 章中所说，Solr 的反向索引性质可以立即返回分面总数。Solr 允许用户在字段、排序或任意的查询（包括本章中的所有函数或用户自定义的函数）上分面，这使得 Solr 的实时分析计算的范围巨大。向组合中添加分面透视可以实现跨多个索引集的连接查询，而且还可以折叠（分组）结果分面，因此你会发现 Solr 将是一个很好的实时分析引擎。

哪些情况可以使用 Solr 作为数据分析引擎？举一个例子，一个网站可以将其服务器日志或访客行为信息发送到 Solr，这样就可以进行基于特定日期、关键字或用户的查询，并可以绘制数据点（网页浏览）来匹配任意查询的数据总数。求职网站可以绘制工作或发布的简历增长数、特殊技能的增长趋势，以及技能、位置、教育水平或经验任意组合的薪酬水平的增长。电子商务网站可以绘制随时间推移的购买特定类别产品的增长趋势，及时挖掘相关属性，例如价格敏感度和区域收益。

Solr 在多字段和数十亿个数据点的规模水平上能够快速处理任意复杂的布尔查询，因此许多组织都选择 Solr 作为实时大数据分析产品的传输装置，这赋予了客户用于对数据进行切片和切块来洞察数据的权利。利用本章涉及的复杂查询操作的知识 and 第 8 章所讲的分面知识，你就可以使用 Solr 快速开发自己的分析引擎，来从数据中获得情报。

## 15.7 本章小结

本章介绍了 Solr 的许多更复杂的数据操作，包括查询函数以及如何轻松地插入自定义函数插件。同时，也看到了 Solr 强大的地理空间搜索功能，基于经度和纬度的基础半径搜索，以及索引搜索任意复杂形状。

本章还介绍了具有多层分面嵌套能力的分面透视，如果每一个更高层级的约束被应用，则每个级别代表着返回的计数。分面的函数功能和能力展现了 Solr 丰富的数据分析能力。

除了自定义函数，本章还介绍了在查询时将外部数据集成到文档中的两种方式：ExternalFileFields 和连接查询。ExternalFileFields 使得用户可以定义在 Solr 索引之外的字段的数据查询函数，连接查询让用户能够匹配基于不同查询的其他文档值，无论是同一 Solr 索引的文档，还是相同 Solr 内核的文档。本章介绍的功能支持用户执行 Solr 中优秀的复杂查询和分析操作，远胜于传统的关键字匹配和相关度排名。当然还有许多方法让传统的匹配和相关性排名更强大，下一章将深入讨论这些方法。



# 16 精通相关度

## 本章要点

- 调试相关度得分
- 根据字段、词项和负载提升相关度
- 使用函数查询改进相关度得分
- 使用 Solr 进行个性化搜索与推荐
- 开展相关度实验及评测

第3章简要介绍了 Solr 的默认相关度评分算法，即如何计算查询及其匹配的文档之间（默认）的相似度得分。随后的章节里介绍了许多搜索功能，例如，扩展 Solr、改善用户体验、配置文本以及查询分析，目的是为了找到与查询相匹配的文档集。

找到相关的文档集之后，需要根据相关度对文档进行排序，以确保搜索用户可以在搜索结果的顶部找到最佳匹配的文档。尽管 Solr 的默认相似度计算适用于普通文本，通过传入有关搜索内容的其他信息也可以明显改进搜索结果的相关度。

如果明确了文档中哪些字段（例如，标题字段）比较重要，Solr 就会在计算相关度得分时提升这些字段的权重。此外，如果文档包含用户行为数据，例如，点击数或购买数，那么可以将这些信息提交给 Solr，它可以根据相似用户的共同行为来判断文档之间的相关性。这是许多推荐引擎的基本工作原理，可以使用这些相同的



技术构建个性化搜索体验或基于 Solr 的推荐引擎。

本章会给出许多提示和技巧，包括以下内容：在 Solr 中替换相似度计算方法，根据函数提升相关度，根据用户行为数据对文档相关度进行增减，测试相关度试验的结果等。虽然 Solr 提供了不错的、开箱即用的相关度评分方法，但是改进搜索结果的相关度会让用户尽可能快速地找到他们需要的内容。这种改进能让客户开心，搜索企业也会获得更多价值回报。这是一个具有挑战性的问题。由于相关度对任何业务都会带来可测度的影响，多数成熟的搜索企业会持续不断地进行相关度调整。本章介绍如何让搜索系统提供更多相关的搜索结果，并在使用 Solr 的过程中获得最大价值。

## 16.1 相关度调整的影响

查询结果的相关度是搜索引擎与其他数据存储方案之间最重要的差异之一。数据库能够查询数十亿文档，快速地检索它们，但在返回的大量数据中不能指明哪些文档是最相关的，这是令人沮丧的用户体验。由于 Solr 为关键词搜索提供了不错的、开箱即用的相关度算法（参见 3.2 节的相关度算法介绍），可能有人会问，对 Solr 搜索的相关度进行微调或其他试验会带来什么好处？

这个问题的答案在很大程度上取决于，所谓的“相关度”到底指什么？第 3 章介绍过查准率与查全率。回忆一下，查准率回答了“哪些文档是应该被返回的文档？”，而查全率回答了“返回的文档是所有应该被返回的文档吗？”。虽然有可能做到同时提高查准率与查全率，但两者之间存在固有的互逆关系，这就使得大多数搜索应用必须对两者做出权衡考虑。

如果唯一的目标是返回全部的正确结果，你可以达到 100% 的查全率，但这样做并不一定能打动用户。同样地，如果唯一的目标是结果正确，通过返回最佳的匹配结果可以达到接近 100% 的查准率（可能偶尔会出错，但希望不是经常犯错）。再强调一次，如果用户只想看到最相关的搜索结果，却让他们尝试浏览多个可能的相关文档，这样的话，不会给他们留下什么深刻印象。

如果用户使用关键字 hamburger 搜索餐馆，结果 Burger King（汉堡王）的多个店面占据了搜索结果的第一页，这些店面都是相关的搜索结果吗？从某种意义上说，它们确实是相关的：汉堡王的所有店面与关键词 hamburger 是等价相关的。然而，从用户的角度看，他们有时会考虑哪个餐馆的地理位置更近一些，或者希望找到不同种类的餐馆，而不是一家餐馆的多个分店。

由于用户考虑很多因素，在用户判断是否相关的认知中，关键词的相关度显然只是影响因素之一。例如，对于通用网络搜索引擎，最终用户的相关度认知可能是唯一考虑指标；对于某些搜索应用，如电子商务网站，从搜索中获得回报可能是更

重要的考虑指标；对于求职搜索引擎，求职搜索的职位总数可能是首要目标。对于相亲网站，约会的匹配数（或者第二轮约会的客户数）可能是黄金指标。在这些搜索应用中，对有些搜索而言，为每个查询找到一个最佳答案是很重要的；对其他一些搜索而言，在用户之间平均分配搜索结果是很重要的。谁都不希望看到这样的情况：求职网站上每个人都在申请同一个职位；相亲网站上很多人与同一个人约会。

准确地把握搜索应用的所属领域，这一点很重要，这样才能更好地改进搜索的相关度。毫无疑问，如果对搜索应用的相关度进行优化，不管是否进行定量评测，搜索应用迈向成功的步伐就已经切切实实地跨出了一大步。

本章会介绍多种相关度改进方法，包括关键词相关度评分的核心相似度算法，根据地理接近度、内容新鲜度、文档流行度来提升相关度，以及基于用户行为的集体智慧来汇聚相关的文档等。你将学习如何改进搜索应用的相关度以及如何评价改进效果。首先，我们介绍 Solr 相关度计算的调试方法，以及如何根据发现的问题对查询进行初步调整。

## 16.2 相关度计算的调试

在介绍 Solr 相关度评分的改进方法之前，我们先熟悉一个有用的内置工具，它用于调试查询匹配文档的相关度得分。在前面章节中，你已经见到过搜索结果中每个文档的最终得分，虽然第 3 章已经介绍了相关度得分的计算原理，但还没有对相关度得分的计算过程进行逐行调试。

要从 Solr 中取回这些调试信息，需要在 Solr 请求中设置 `debug=results` 或 `debug=true` 参数，以启用查询调试。`debug` 参数出现在 Solr 响应的末尾部分，以单独一节返回调试信息。

在调试相关度计算之前，我们首先准备一个搜索示例，并向 Solr 发送一些文档。与前面章节的做法类似，配置好示例内核并启动 Solr：

```
cd $SOLR_INSTALL/example/  
cp -r $SOLR_IN_ACTION/example-docs/ch16/cores/ solr/  
java -jar start.jar
```

Solr 运行起来之后，我们向 Solr 发送几个示例文档：

```
cd $SOLR_IN_ACTION/example-docs/  
java -Durl=http://localhost:8983/solr/no-title-boost/update  
  -jar post.jar ch16/documents/no-title-boost.xml
```

向 Solr 提交的文档如下所示：

```

<add>
  <doc>
    <field name="id">1</field>
    <field name="restaurant_name">Red Lobster</field>
    <field name="description">
      We deliver the freshest caught seafood every day.
    </field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="restaurant_name">Joe's Crab Shack</field>
    <field name="description">
      We serve delicious red crabs, large lobsters, and other delicious
      seafood. Our lobsters are our specialty.
    </field>
  </doc>
</add>

```

为了演示如何对 Solr 的相关度计算进行调试，我们向 Solr 提交一条查询，查找两个文档中都出现的词项：red lobster。代码清单 16.1 是 debug=true 的查询调试输出结果。

#### 代码清单 16.1 搜索结果相关度解释的调试输出示例

##### 查询请求

```

http://localhost:8983/solr/no-title-boost/select?
defType=edismax&
q=red lobster&
qf=restaurant_name description&
debug=true&

```

##### 搜索结果

```

{
  ...
  "response": {"numFound": 2, "start": 0, "docs": [
    {
      "id": "2",
      "restaurant_name": "Joe's Crab Shack",
      "description": "We serve delicious red crabs, large lobsters, and
other delicious seafood. Our lobsters are our specialty."},
    {
      "id": "1",
      "restaurant_name": "Red Lobster",
      "description": "We deliver the freshest caught seafood every day."}]
  },
  "debug": {
    "rawquerystring": "red lobster",
    "querystring": "red lobster",
    "parsedquery": "(+ (DisjunctionMaxQuery((description:red |
restaurant_name:red)) DisjunctionMaxQuery((description:lobster |
restaurant_name:lobster))))/no_coord",
    "parsedquery toString": "(+((description:red | restaurant name:red)

```

```

(description:lobster | restaurant_name:lobster))",
  "explain":{
    "2":
1.7071067 = (MATCH) sum of:
  0.70710677 = (MATCH) max of:
    0.70710677 = (MATCH) weight(description:red in 1) [DefaultSimilarity],
    result of:
      0.70710677 = score(doc=1,freq=1.0 = termFreq=1.0
), product of:
  0.70710677 = queryWeight, product of:
    1.0 = idf(docFreq=1, maxDocs=2)
    0.70710677 = queryNorm
  1.0 = fieldWeight in 1, product of:
    1.0 = tf(freq=1.0), with freq of:
      1.0 = termFreq=1.0
    1.0 = idf(docFreq=1, maxDocs=2)
    1.0 = fieldNorm(doc=1)
0.99999994 = (MATCH) max of:
  0.99999994 = (MATCH) weight(description:lobster in 1)
  [DefaultSimilarity], result of:
    0.99999994 = score(doc=1,freq=2.0 = termFreq=2.0
), product of:
  0.70710677 = queryWeight, product of:
    1.0 = idf(docFreq=1, maxDocs=2)
    0.70710677 = queryNorm
  1.4142135 = fieldWeight in 1, product of:
    1.4142135 = tf(freq=2.0), with freq of:
      2.0 = termFreq=2.0
    1.0 = idf(docFreq=1, maxDocs=2)
    1.0 = fieldNorm(doc=1),
"1":
0.8838835 = (MATCH) sum of:
  0.44194174 = (MATCH) max of:
    0.44194174 = (MATCH) weight(restaurant_name:red in 0)
    [DefaultSimilarity], result of:
      0.44194174 = score(doc=0,freq=1.0 = termFreq=1.0
), product of:
  0.70710677 = queryWeight, product of:
    1.0 = idf(docFreq=1, maxDocs=2)
    0.70710677 = queryNorm
  0.625 = fieldWeight in 0, product of:
    1.0 = tf(freq=1.0), with freq of:
      1.0 = termFreq=1.0
    1.0 = idf(docFreq=1, maxDocs=2)
    0.625 = fieldNorm(doc=0)
0.44194174 = (MATCH) max of:
  0.44194174 = (MATCH) weight(restaurant_name:lobster in 0)
  [DefaultSimilarity], result of:
    0.44194174 = score(doc=0,freq=1.0 = termFreq=1.0
), product of:
  0.70710677 = queryWeight, product of:
    1.0 = idf(docFreq=1, maxDocs=2)
    0.70710677 = queryNorm
  0.625 = fieldWeight in 0, product of:

```

```

1.0 = tf(freq=1.0), with freq of:
  1.0 = termFreq=1.0
1.0 = idf(docFreq=1, maxDocs=2)
0.625 = fieldNorm(doc=0)}},
...
"timing":{
  "time":4.0,
  "prepare":{
    "time":0.0,
    "query":{
      "time":0.0},
    "facet":{
      "time":0.0},
  },
  "process":{
    "time":3.0,
    "query":{
      "time":0.0},
    "facet":{
      "time":0.0},
  },
}
... }
}}

```

如上所示，代码清单 16.1 返回了一个完整的调试区块，其中包括已解析的查询、每个搜索组件的耗时、与查询匹配的每个文档的完整相关度计算的解释内容。每一部分内容对调试搜索性能都非常有用。如果想要查看相关度计算的完整解释，使用 `debug=results`（而不是 `debug=true`）参数来去除其他调试内容。另外，如果想要按照字段查看每个文档的相关度解释，可以通过字段列表（`fl`）参数请求对应字段的相关度解释，如代码清单 16.2 所示。

#### 代码清单 16.2 按照字段返回每个文档的相关度解释

##### 查询请求

```

http://localhost:8983/solr/no-title-boost/select?
defType=edismax&
q=red lobster&
qf=restaurant_name description&
fl=id,restaurant_name,description,[explain]

```

请求每个文档的每个  
字段的相关度解释。

##### 搜索结果

```

{
  ...
  "response":{"numFound":2,"start":0,"docs":[
    {
      "id":"2",
      "restaurant_name":"Joe's Crab Shack",
      "description":"We serve delicious red crabs, large lobsters, and
other delicious seafood. Our lobsters are our specialty.",
      "[explain]":""
    }
  ]}
}

```

```

1.7071067 = (MATCH) sum of:
  0.70710677 = (MATCH) max of:
    0.70710677 = (MATCH) weight(description:red in 1) [DefaultSimilarity],
      result of:
        0.70710677 = score(doc=1,freq=1.0 = termFreq=1.0
), product of:
    0.70710677 = queryWeight, product of:
      1.0 = idf(docFreq=1, maxDocs=2)
      0.70710677 = queryNorm
    1.0 = fieldWeight in 1, product of:
      1.0 = tf(freq=1.0), with freq of:
        1.0 = termFreq=1.0
      1.0 = idf(docFreq=1, maxDocs=2)
      1.0 = fieldNorm(doc=1)
0.99999994 = (MATCH) max of:
  0.99999994 = (MATCH) weight(description:lobster in 1)
    [DefaultSimilarity], result of:
      0.99999994 = score(doc=1,freq=2.0 = termFreq=2.0
), product of:
    0.70710677 = queryWeight, product of:
      1.0 = idf(docFreq=1, maxDocs=2)
      0.70710677 = queryNorm
    1.4142135 = fieldWeight in 1, product of:
      1.4142135 = tf(freq=2.0), with freq of:
        2.0 = termFreq=2.0
      1.0 = idf(docFreq=1, maxDocs=2)
      1.0 = fieldNorm(doc=1)"
},
{
  "id":"1",
  "restaurant_name":"Red Lobster",
  "description":"We deliver the freshest caught seafood every day."},
  "explain":":
0.8838835 = (MATCH) sum of:
  0.44194174 = (MATCH) max of:
    0.44194174 = (MATCH) weight(restaurant_name:red in 0)
      [DefaultSimilarity], result of:
        0.44194174 = score(doc=0,freq=1.0 = termFreq=1.0
), product of:
    0.70710677 = queryWeight, product of:
      1.0 = idf(docFreq=1, maxDocs=2)
      0.70710677 = queryNorm
    0.625 = fieldWeight in 0, product of:
      1.0 = tf(freq=1.0), with freq of:
        1.0 = termFreq=1.0
      1.0 = idf(docFreq=1, maxDocs=2)
      0.625 = fieldNorm(doc=0)
    0.44194174 = (MATCH) max of:
      0.44194174 = (MATCH) weight(restaurant_name:lobster in 0)
        [DefaultSimilarity], result of:
          0.44194174 = score(doc=0,freq=1.0 = termFreq=1.0
), product of:
            0.70710677 = queryWeight, product of:

```

计算文档 2 的 description 字段中词项 red 的得分。

在文档 2 中, red 在 decription 字段出现 1 次 (termFreq=1.0)。

计算文档 2 的 description 字段中词项 lobster 的得分。

在文档 2 中, lobster 在 decription 字段出现 2 次 (termFreq=2.0)。

计算文档 1 的 restaurant\_name 字段中词项 red 的得分。

在文档 1 中, red 在 restaurant\_name 字段出现 1 次 (termFreq=1.0)。

计算文档 1 的 restaurant\_name 字段中词项 lobster 的得分。

```

1.0 = idf(docFreq=1, maxDocs=2)
0.70710677 = queryNorm
0.625 = fieldWeight in 0, product of:
1.0 = tf(freq=1.0), with freq of:
    1.0 = termFreq=1.0
1.0 = idf(docFreq=1, maxDocs=2)
0.625 = fieldNorm(doc=0)"
1}}

```

在文档 1 中, lobster 在 restaurant\_name 字段出现 1 次 (termFreq=1.0)。

每个词项的 idf 值为 1.0, 此处不影响得分。

在本章中, 我们会忽略代码清单 16.1 中的其他调试信息, 只关注代码清单 16.2 对每个文档的解释部分。如果你被相关度计算的大量解释信息搞懵的话, 可以参阅 3.2 节, 那里逐行详细介绍了默认的相关度计算细节。第 3 章介绍过 `tf(termFreq)`、`idf` 及相关度的其他许多变量。理解所有的代码可能会多花一些时间, 加上再回顾一下 3.2 节的内容, 这样会在调试 Solr 搜索应用时能够更好地理解相关度的解释信息。

在代码清单 16.2 的例子中, 让搜索用户感到烦扰的一个问题是, 搜索 red lobster 时, 一家名为 “Joe’s Crab Shack” 的餐馆比实际名为 “Red Lobster” 的餐馆的排名更靠前。参照代码清单 16.2 中对相关度的解释, Joe’s Crab Shack 的总得分是 1.7071067, 即词项 red 得分 (score=0.70710677) 与词项 lobster 得分 (score=0.99999994) 的总和。那家名为 Red Lobster 的餐馆的相关度得分是 0.8838835, 即词项 red 得分 (score=0.44194174) 和词项 lobster 得分 (score=0.44194174) 的总和。在这个例子中, 每个词项的 idf 值都是 1.0, 因此不会影响结果。

两个得分的差异是如何造成的? 从相关度解释的分析看, 其中似乎存在两个主要因素。

- 在 Joe’s Crab Shack 文档中, 词项 lobster 出现两次, 所以它的 `tf(termFreq)` 值翻番 (即 2 倍得分)。
- 将 `description` 字段的 `omitNorms` 参数设置为 `true`, `restaurant_name` 字段的 `omitNorms` 参数设置为 `false`。相对于 `description` 字段, 这样做会对 `restaurant_name` 字段不公平, 因为 `restaurant_name` 字段本身的内容较少, 内容的相对长度会被忽略。

了解了这两个变量之后就可以做一些修改, 在搜索餐馆名称时, 让 Red Lobster 文档在搜索结果中排名靠前。由于不太可能修改内容本身, 所以需要考虑对 `description` 字段开启规范化, 以避免较长的文本对 `restaurant_name` 字段造成不公平。因为 `restaurant_name` 字段明显比 `description` 字段更具相关性, 所以可以考虑为 `restaurant_name` 字段增加权重。下一节讨论影响内容相关度的各种提升方法。



## 16.3 提升相关度

上一节介绍了如何调试 Solr 的相关度得分，如果发现相关度得分的计算方法存在问题时，那该怎么办？所幸，Solr 提供了相关度计算的多种调整方法，让我们可以轻松地修改词项、字段，甚至整个文档的权重。多数调整方法既可用于索引阶段，也可用于查询阶段，甚至可以在查询级别有选择性地对文档进行重排序。本节介绍如何基于对内容重要特征的理解来调整相关度。

### 16.3.1 字段提升

在大多数文档中，某些字段会被认为比其他字段更具相关性。例如，在产品字段中，`production_name` 字段会比 `description` 字段更具相关性。这是因为，客户最有可能搜索产品的名称。因此如果能匹配到精确的产品名称，应首先考虑返回精确的名称匹配结果。类似地，如果文档是有关社交网络的个人信息，与包含朋友列表的字段相比，`person_name` 字段更重要一些，精确的人名匹配应该显示在搜索结果的顶部。

#### 索引阶段的字段提升

在 Solr 中，指定某些字段比其他字段更重要的方法不止一种。如果在索引时就知道某个字段更重要，而且该字段的重要性不会改变，那么可以在提交文档给 Solr 时使用字段级别的提升。这里为 16.2 节的文档设置如下的提升值：

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="restaurant_name" boost="10.0">Red Lobster</field>
    <field name="description">
      We deliver the freshest caught seafood every day.
    </field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="restaurant_name" boost="10.0">Joe's Crab Shack</field>
    <field name="description">
      We serve delicious red crabs, large lobsters, and other delicious
      seafood. Our lobsters are our specialty.
    </field>
  </doc>
</add>
```

你可以直接使用随书附带的源代码，通过以下命令向 Solr 提交这些文档：

```
cd $SOLR_IN_ACTION/example-docs/
java -Durl=http://localhost:8983/solr/title-boost/update
➡ -jar post.jar chl6/documents/title-boost.xml
```

从这个例子可以看出，这两个文档的 `restaurant_name` 字段都包含了一个添加上去的 `boost=10.0` 属性。这意味着，在搜索多个字段时，该字段的重要性是其他字段的 10 倍。将 `restaurant_name` 字段的因子提升为 10，以确保类似 Joe's Crab Snack 餐馆描述中的关键词不能轻易超越其他餐馆的 `restaurant_name` 字段的相同关键词，正如之前的 `red lobster` 查询中出现的情况。

当 `restaurant_name` 字段得到提升的文档被 Solr 索引以后，重新执行之前的 `red lobster` 查询。代码清单 16.3 显示了该查询及其搜索结果。

### 代码清单 16.3 索引阶段提升一个重要字段之后的搜索结果改进

#### 查询请求

```
http://localhost:8983/solr/title-boost/select?
defType=edismax&
q=red lobster&
qf=restaurant_name description&
fl=id,restaurant_name,description
```

#### 搜索结果

```
{
  ...
  "response": { "numFound": 2, "start": 0, "docs": [
    {
      "id": "1",
      "restaurant_name": "Red Lobster",
      "description": "We deliver the freshest caught seafood every day." },
    {
      "id": "2",
      "restaurant_name": "Joe's Crab Shack",
      "description": "We serve delicious red crabs, large lobsters, and
        other delicious seafood. Our lobsters are our specialty." }
  ] }
}
```

由于在索引阶段调整了 `restaurant_name` 字段的权重，Red Lobster 目前排名比之前提高了。

由于 `description` 字段没有提升权重，所以该文档得分依然相对较低。

这样处理之后，`red Lobster` 餐馆的文档在搜索结果的排名提前了。如果在该查询的 URL 后面添加 `debug=results`，就可以了解在索引阶段提升字段后，相关度计算的变化情况（与代码清单 16.2 相比）。此外，通常认为结果匹配中餐馆名称比 `description` 字段更具相关性，这样的处理方式解决了搜索索引中未来很多文档会遇到的相同问题。

虽然索引阶段的字段提升是有效的，但这样也会带来一些问题。如果想要通过调整某个字段的 `boost` 值来观察其如何影响搜索的相关度，这样会发生什么情况呢？

要在索引阶段提升字段，就需要重新对所有文档进行索引。此外，索引阶段的字段 `boost` 值存储在 `fieldNorm` 中（参见 3.2.5 节）。`fieldNorm` 会把字段 `boost` 值与文档 `boost` 值以及字段长度规范一起压缩成一个字节。正因为如此，要在索引阶段进行提升，需要在字段上设置 `omitNorms=false`（否则 `boost` 值不会被存储），而且由于 `boost` 值与其他值一起被压缩成一个字节，提升量可能不准确。由于存在这些限制，大多数情况下在查询阶段提升字段会是更好的选择。

### 查询阶段的字段提升

从理论上讲，索引阶段的字段提升是有意义的，但现实中存在很多限制。3.2.4 节介绍过一个更简单的方法，即在查询阶段提升字段。它可以实现同样的目的，而且允许灵活地动态修改字段 `boost` 值。

### 人工查询

```
http://localhost:8983/solr/no-title-boost/select?
q=restaurant_name:(red lobster)^10 OR description:(red lobster)
```

### eDisMax

```
http://localhost:8983/solr/no-title-boost/select?
defType=edismax&
q=red lobster&
qf=restaurant_name^10 description
```

在这个例子中，我们再次设置 `restaurant_name` 字段的 `boost` 值是 `description` 字段的 10 倍，并取消索引阶段的提升。`schema.xml` 的字段设置 `omitNorms=false` 就没有必要了，不需要在索引阶段对字段的 `boost` 值进行存储，那么索引阶段存储的 `boost` 值失真问题也就不复存在了。总体来说，除非需要为同一字段的不同文档分别设置 `boost` 值，否则只需在查询阶段（而不在索引阶段）进行字段提升。

## 16.3.2 词项提升

在某些情况中，为字段的所有词项统一设置 `boost` 值，倒不如为查询的各个词项单独设置 `boost` 值。3.2.4 节曾介绍过有关内容，词项提升与查询阶段的字段提升语法相同，只是将其直接应用在各个词项上：

```
http://localhost:8983/solr/no-title-boost/select?
q=restaurant_name:(red^2 lobster^8)
OR description:(red^2 lobster^8)
```

从上面的查询可以看出，词项 `red` 和 `lobster` 各自拥有一个 `boost` 值。这些

值是根据传入 Solr 的查询词项的相对重要程度得出的。另外，查询阶段词项级别的 boost 值可以与字段级别的 boost 值进行组合，举例如下：

### 人工查询

```
http://localhost:8983/solr/no-title-boost/select?
q=restaurant_name:(red^2 lobster^8)^10
OR description:(red^2 lobster^8)
```

### eDisMax

```
http://localhost:8983/solr/no-title-boost/select?
defType=edismax&
q=red^2 lobster^8&
qf=restaurant_name^10 description
```

在上面例子中，字段（restaurant\_name）和词项（red 与 lobster）都得到了提升。问题随之而来，在提升的字段里，提升的词项该如何进行组合？在 Solr 中，组合提升问题的解决办法非常简单，将两者相乘。上例中 restaurant\_name 字段匹配到 red 与 lobster 两个词项，词项 red 得到的 boost 值为  $(2 \times 10) = 20$ ，词项 lobster 的 boost 值为  $(8 \times 10) = 80$ ，比其他评分都要高。此规则适用于 Solr 中所有类型的提升方法。若存在多个 boost 值，则对它们相乘。

## 16.3.3 负载提升

至此，我们讨论了在索引阶段与查询阶段对文档的字段和查询的词项进行提升，那么在一个字段中如何提升各个词项呢？假如在搜索系统中构建一个词性探测器，想要对文档词项中的所有名词进行提升，使其高于其他词项，该如何处理？一种方法是为重要的词类创建单独的字段，为其赋予更高的 boost 值。这种方法对于数量较少的 boost 值可能会有用，如果搜索应用需要为字段的每个词项设置唯一的 boost 值，又该如何处理？

在文档中对词项进行提升的最佳方法是使用 Solr 提供的负载（payload）功能。当 Solr 将文档的每个词项写入到搜索索引时，可以选择性地存储词项的一个字节数组，其包含查询阶段可能有用的其他信息。此字节数组被称为负载，用于文档评分，作用是影响文档中带有负载的词项的相关度得分。

虽然 Solr 官方支持对负载进行索引，如 DelimitedPayloadFilterFactory，但是开箱即用的 Solr 不支持在查询中使用负载来影响相关度。不过在 Solr 中还是有其他方法可以实现这个功能（<https://issues.apache.org/jira/browse/SOLR-1485>）。一些组织自定义了查询解析器和相似度对象，将负载用于相关度评分，要实现这个高级

功能, 需要进行一些开发工作。我们至少知道有这样的解决方法存在, 以应对未来可能遇到此类问题。

### 16.3.4 函数提升

第 15 章介绍了 Solr 的函数。本节介绍如何在相关度评分模型中使用函数计算。首先, 向示例内核添加一些示例文档:

```
cd $SOLR_IN_ACTION/example-docs/  
java -Durl=http://localhost:8983/solr/distance-relevancy/update  
  -jar post.jar ch16/documents/distance-relevancy.xml  
java -Durl=http://localhost:8983/solr/news-relevancy/update  
  -jar post.jar ch16/documents/news-relevancy.xml
```

将函数置于查询中, 函数本身会被视为词项。与文档进行匹配, 该函数会返回一个数值, 即相关度得分。也就是说, 通过向查询中添加函数, 可以操纵 (或替换) Solr 的相关度算法, 修改相关度评分方法, 以满足搜索应用需求。查询举例如下:

```
http://localhost:8983/solr/distance-relevancy/select?  
q=restaurant_name:(Burger King) AND  
_query_:"{!func}recip(geodist(location,37.765,-122.43),1,10,1)"
```

该请求会找到包含 Burger 和 King 两个词项的所有文档, 计算每个文档的相似度得分, 大致等于 `score("Burger")`、`score("King")`、`LocationProximityBoost` 三者相加。

这样做的好处在于, 查询中指定的地理位置靠近的文档会得到额外的相关度提升, `boost` 值最高为 10。与关键词查询相比, 地理位置越靠近的文档在搜索结果中的排名越靠前。通过调整 `recip` 函数的输入, 可以控制距离曲线的斜率和相关度提升的最大值。如果恰巧知道用户的位置或者他们感兴趣的地点, 就可以很容易地在他们的查询请求中添加一个地理位置过滤器。在某些情况下, 我们可能仅想要提升与兴趣点位置相近的文档, 而不是在附近的半径内过滤掉不相关的文档。位置接近的提升可以帮助那些既相关又在附近的文档出现在搜索结果的顶部。

除了基于位置的提升之外, 许多搜索应用还会根据内容的新颖性与流行度对相关度进行调整。我们以一个不间断发布新闻文章的在线新闻网站为例来说明。在新闻网站中, 一篇文章的相关度可能涉及关键词的相关度、文章主题的地理位置、文档多久前发布, 以及文章的流行程度等诸多因素的组合。如果只考虑位置和关键词两个相关度, 可能会显示出陈旧的文章。对新闻网站而言, 这种做法通常是不可取的。如果只按照新闻发布日期排序, 得到的是与用户关键词匹配的模糊结果, 而不是最佳的相关匹配。通过增加流行度这一相关度因素, 可以根据其他网站访问者的浏览趋势来提升文档的排名。提交给 Solr 的这类查询请求会是什么样的呢?

```
http://localhost:8983/solr/news-relevancy/select?
fq={!cache=false v=$keywords}&
q=_query_:"{!func}scale(query($keywords),0,100)"
  AND _query_:"{!func}div(100,map(geodist(location,$pt),0,1,1))"
  AND _query_:"{!func}recip(rord(publicationDate),1,100,1)"
  AND _query_:"{!func}scale(popularity,0,100)"&
keywords="street festival"&
pt=33.748,-84.391
```

过滤器 (fq 参数) 会限制关键词匹配的结果, 上面的关键词是 "street festival"。查询 (q 参数) 依次计算出各个值, 取值区间为 0 ~ 100: (1) 关键词相关度得分, (2) 地理位置接近得分, (3) 文档的时效性, 以及 (4) 文档的流行度。虽然 recip 函数与 div 函数分别限制最高得分为 100, div 函数中的 map 确保 100 不会被划分为小于 1 的值, 但是查询函数与流行度比较难控制, 除非确保已索引的文档被设置在 0 到 100 之间。因此, 在一些情况下, 我们会使用 scale 函数, 找出文档的最小值与最大值, 将文档的所有值按比例缩放到 0 到 100 之间。

参照书中前面的例子, 对关键词相关度得分的调整、地理接近程度的计算、流行程度的调整可能都是合理的, 这里对较新的文档提升重要性可能不那么明显了。由于这是一个新闻网站, 不管最近一篇文章已经发布多久, 我们都认为文章越新越相关。因此, 按照索引中的文档次序, 使用 rord 函数对新文档进行提升。与此同时, 惩罚所有的旧文档 (相对于新文档而言), 即使它们仅仅是几小时或几分钟前发布的。对日期进行提升的大多数搜索应用中, 你可能想要采用不同的提升策略: 基于文档发布的时间, 而不是基于随后发布的新文档数量。要做到这一点, 同时还要把得分保持在 0 到 100 之间, 可以继续使用 scale 函数来实现: scale(ms(publicationDate),0,100)。这两种方法的区别在于: scale(ms(...)) 组合根据时间对新文档的提升较大, 而 recip(rord(...)) 组合在少量较新文档存在时会给予较大提升。通过函数对日期和其他值来影响文档相关度的方法还有很多, 这里只介绍了几种。

通过组合函数, 如 scale、div 与 recip 函数, 每个函数的结果值在 0 到 100 之间, 在理想情况下查询的最高相关度得分是 400, 即存在兼具最相关的关键词、最靠近的地理位置、最新且最热门四者为了一身的文档。Solr 返回最终得分之前对其进行规范化, 规范化的得分会低于原始得分。如果开启调试和显示相关度解释, 缩放后的得分可以显示出来。虽然已经对四个变量依次进行了加权平均, 你也可以自主选择相关度变量, 根据搜索应用的需求, 设置加权集合。还需注意一点, 使用函数会增加查询的复杂度, 这样会影响搜索性能。一些函数 (如 scale) 需要进行额外的其他处理, 将每个文档值纳入考虑范围。你需要谨慎地在查询速度与复杂相关度计算带来的价值之间进行权衡。



### 16.3.5 词项邻近度提升

如果与用户搜索的短语相匹配的文档总是排在包含各个词项的文档的后面，你可能要考虑在查询请求中对词项邻近度进行提升，以改进相关度计算。

执行搜索时，开箱即用的 Solr 将两种信息检索模型——词项的布尔匹配与向量空间相关度计算（相关度模型参见 3.2 一节）——结合使用。由于向量的相关度得分是每个词项得分的总和，所以默认情况下，查询的词项邻近程度信息并未考虑在内。这意味着，如果搜索 `content:(statue of liberty)`，只要包含 `statue`、`of` 和 `liberty` 三个词项的文档就会被匹配出来，而包含精确短语 `"statue of liberty"` 的文档得分低于包含三个独立词项的文档得分。

在大多数搜索应用中，对词项更接近的那些文档进行提升是有益的做法。在 Solr 中通常有两种实现方法：一种是使用 `eDisMax` 查询解析器，另一种是使用默认的 `Lucene` 查询解析器。

#### 使用 eDisMax 查询解析器提升词项邻近度

如果使用 `eDisMax` 查询分析器（参见第 7 章），在短语相关度提升中可以传递一些特殊的参数。这些参数主要是 `pf`（短语字段）参数，它能对字段中更靠近的词项进行额外提升。`pf` 参数的语法是 `field~slop^boost`。如果在一个 Solr 内核的 `content` 字段上搜索 `big`、`data`、`analytics` 这三个词，想要对三个词彼此邻近的情况进行相关度提升，Solr 的查询写法如下所示：

```
/select?
defType=edismax&
q=big data analytics&
qf=content&
pf=content~3^10
```

除了 `pf` 参数，`eDisMax` 查询解析器为短语字段定义提供其他字段，包括 `bigrams`（`pf2` 参数）与 `trigrams`（`pf3` 参数）。如果之前的查询指定参数为 `pf2=content`，那么只要 `content` 字段中出现短语 `"big data"` 或 `"data analytics"`，就会对其进行提升。要指定 `bigram` 短语匹配的精确程度，设置相应的双精度 `ps2` 参数（其表示短语的 `slop` 值为二元）。`eDisMax` 查询解析器也支持三元参数，`pf3` 参数对一个字段设置相应的短语 `slop` 值。当文档包含三个邻近词项时——就对其进行提升，在这个例子中 `"big data analytics"` 属于这种情况。短语字段和短语的 `slop` 参数的有关内容，请参见 7.5.4 一节。通过设置这些参数，可以找到邻近词项搜索的文档相关度改进方法。



## 使用 eDisMax 查询解析器提升词项邻近度

如果不使用 eDisMax 查询解析器,而是用传统的 Lucene 查询解析器,那么做法略有不同。如果不使用不同的请求参数,可以在查询 `q` 参数中将邻近度 `boost` 值添加为一个词项。这样做的 Lucene 语法是 `"term1 term2 ... termN"~slop^boost`。举例来说, `"customer service"~2^10` 查询会匹配包含 `customer` 和 `service` 两个词项且词项间距不超过 2 个位置的文档,并将其 `boost` 值提升 10 倍。很多情况下,我们希望匹配包含 `customer` 和 `service` 两个词项的所有文档,但只对词项邻近度高的文档进行提升。下面的查询示例满足这个要求:

```
1 q=+customer +service OR "customer service"
2 q=+customer +service OR "customer service"~2^10
3 q=customer OR service +{some other terms}
  +(*:* OR "customer service"^100)
4 q=+customer +service +representative
  OR "customer service representative"~10000^10
```

在第一个查询例子中,文档只要出现 `customer` 和 `service` 两个词项,就会被匹配。但是如果两个词项彼此相邻,是一个短语,那么它们会被视为一个额外的匹配,会被赋予双倍的相关度提升(因为每个词项都匹配了 2 次)。在这个例子中, `"customer service"` 精确短语查询隐含了一个 `boost` 为 1 的提升值,所以它等价于 `"customer service"^1` 或 `"customer service"~0^1`。

在第二个查询例子中, `"customer service"~2` 会匹配包含 `customer` 和 `service` 两个词项且位置间隔在 2 以内的文档,并对其提升相关度。因此,这个查询会匹配 `"customer needs service"` (间距为 1)、`"service customer"` (间距为 2)、`"customer needs some service"` (间距为 2),但不会匹配 `"customer really needs some service"` (间距为 3) 或 `"service the customer"` (间距为 3)。这个例子的相关度 `boost` 值为 10,表示邻近查询的相关度提升 10 倍。

第三个查询示例演示了如何在查询中仅对词项进行邻近度提升。在这个例子中,为了避免词项邻近度提升对搜索结果数量造成影响,我们将查询中整个邻近度提升作为请求选项,让它匹配所有文档。在不影响搜索结果数量的情况下,可以进行相关度提升。

第四个查询有两点需要解释。其一,邻近查询中添加了不少两个词项。位置偏移计算会考虑每个词项的情况,因此词项多时,最好设置 `slop` 变量。其二,这个邻近查询的 `slop` 值非常大,达到 10 000。这可能看起来很可笑,它实际等价于搜索 `+customer +service +representative`,所以词项间隔上限为 1000,这有

可能比文档本身的长度要长。因此，这两个查询效果是一样的。词项之间越接近，邻近查询的得分越高。由于此例中邻近查询的 `slop` 值较大，实际上是根据词项的接近程度对文档进行提升。由于 Solr 的默认相关度算法不包括邻近度提升，因此在查询中增加邻近度提升会改进搜索应用的相关度。邻近查询的提升需要额外的处理时间，所以要谨慎地在改进相关度与影响查询速度之间做出权衡。

### 16.3.6 提升重要文档的相关度

在搜索应用中，提升相对重要的文档的相关度主要有三种方法。最简单的方法是在索引阶段进行提升。做法与 16.3.1 节介绍的在索引阶段进行字段级别的提升是一样的，只不过应用于整个文档，而不是单个字段：

```
<add>
  <doc boost="10.0">
    <field name="id">1</field>
    <field name="restaurant_name">Red Lobster</field>
    <field name="description">We deliver the freshest caught
      seafood every day.</field>
  </doc>
</add>
```

从内部看，文档级别的提升相当于在索引阶段对文档所有字段进行字段级别的提升。索引阶段文档级别的提升仅是提供了一种便利的方法，方便对文档所有字段在索引阶段进行提升。这也意味着，这种提升不适用于 `omitNorms` 为 `True` 的字段搜索。之前讨论过的索引阶段字段级别提升的优缺点在这里同样适用。

16.3.1 节推荐使用查询阶段的字段提升替换索引阶段的字段提升，你可能想知道在查询阶段是否也有对应的文档级别的提升？这种提升确实存在，而且之前已经简要介绍过了。

#### 流行度字段的文档相关度提升

为了解决索引阶段文档级别提升的限制，更灵活的文档提升方法是对流行度字段进行索引，在该字段上执行函数查询，以实现文档流行度的提升。16.3.4 节已经介绍过这个方法，通过已索引的流行度字段来尝试提升新闻报道的相关度。从技术上讲，索引阶段的提升是将每个词项的相关度相乘，这与流行度字段的函数查询有所不同，后者是计入整体得分总和。尽管通过构造函数查询可以模仿索引阶段的提升，但在实践中，通过添加 `boost` 值就可以实现想要的输出结果。因此，如果添加 `boost` 值没有导致问题出现，过分关注这方面的细节可能不是一种成熟的优化选择。

索引阶段的文档提升和通过流行度字段的函数查询来提升文档，这两种方法关注的都是在全局范围内文档相关度的相对提升。对电子商务应用而言，某些商品往

往卖得更好；对新闻网站而言，一些热门新闻报道引领了关注趋势。在许多搜索应用中，可能只希望基于特定查询来提升文档，而不是对所有查询进行全面提升。

在代码清单 16.1 的 Red Lobster 与 Joes Crab Shack 例子中，我们不是希望包含 Red Lobster 的文档在任何时候要比包含 Joe's Crab Shack 的文档更相关，而是希望对于 red lobster 这个特定查询而言，包含 Red Lobster 显示在前，但包含 Joe's Crab Shack 的文档却显示在了前面。

Solr 针对特定查询提供一个搜索组件，可以将某些指定文档移到搜索结果的顶部，或移除它们。这个组件称为查询提升组件。当默认的相似度计算不能很好地返回用户期望的结果时，查询提升组件可以对这种问题查询进行修正。

### 使用查询提升组件修正个别相关度

为使用查询提升组件，需要在 solrconfig.xml 中添加以下代码来启用它：

```
<searchComponent name="elevation"
  class="solr.QueryElevationComponent" >
  <str name="queryFieldType">text</str>
  <str name="config-file">elevate.xml</str>
</searchComponent>
```

这里需要注意，queryFieldType 类应根据用户查询在 config-file (elevate.xml) 中的解析方式来定义。例如，如果只需要精确匹配用户输入的文本，你可能会考虑使用 string 字段类型。如果希望对用户的查询进行分词，匹配用户输入的所有关键词，那么你可能会考虑使用 text 字段类型，如示例配置所示。定义好查询提升组件之后，还需要执行一个附加步骤。虽然最常见的搜索组件已经默认配置好了 (query、facet、mlt、highlight、stats 及 debug)，但查询提升组件在默认情况下没有开启，你需要在请求处理器中添加它。在这个例子中，我们将创建一个请求处理器映射：

```
<requestHandler name="/elevate" class="solr.SearchHandler" >
  ...
  <arr name="last-components">
    <str>elevation</str>
  </arr>
</requestHandler>
```

配置好查询提升组件之后，最后一个步骤是定义一些查询及对应的文档。针对这些查询，对应的文档要么返回在搜索结果的顶部，要么完全排除在外。elevate.xml 文件应放置在 Solr 内核的 conf/ 目录下，如代码清单 16.4 所示。

## 代码清单 16.4 查询提升组件的 elevate.xml 示例

```

<elevate>
  <query text="red lobster">
    <doc id="1" />
    <doc id="2" exclude="true" />
  </query>
  <query text="some other query">
    <doc id="2" />
    <doc id="3" />
    <doc id="1" />
  </query>
</elevate>

```

对于将该查询，总是将这个文档放在第一位。

不显示这个文档。

一个查询中可以定义多个优先考虑的文档。

启用查询提升组件并定义了提升查询之后，接下来需要为这些查询修改 Solr 的相关度算法。Solr 的内核已经包含了 /elevate 处理器的配置，用以解决我们之前发现的问题，即 Joe's Crab Shack 出现在 "red lobster" 查询的搜索结果顶部。现在回到 Solr 的内核，对原始查询与使用了提升组件的查询进行比较：

## 原始查询

```

http://localhost:8983/solr/no-title-boost/select?defType=edismax&
q=red lobster&qf=restaurant_name description&fl=id,restaurant_name

```

## 原始结果

```

...
"docs": [{
  "id": "2",
  "restaurant_name": "Joe's Crab Shack"},
  {
    "id": "1",
    "restaurant_name": "Red Lobster"}]

```

## 提升查询

```

http://localhost:8983/solr/no-title-boost/elevate?defType=edismax&
q=red lobster&qf=restaurant_name description&fl=id,restaurant_name

```

## 提升结果

```

...
"docs": [{
  "id": "1",
  "restaurant_name": "Red Lobster"}]

```

手动定义查询的搜索结果是耗费人力的工作。因此，这种方法通常只建议用于热门查询的个别修正。你也可以根据外部的可用信息（例如电子商务应用中特定查询的热门购物），为热门查询设置一个预先编译的结果列表，这样做会从用户那里获得更多附加价值。

至此，你已经了解了如何通过文档、词项、负载、词项邻近度及函数对搜索应用的相关度进行显式干预，下一节会介绍更复杂的改进方法：完全替换 Solr 内核的相关度算法。

## 16.4 可插拔的相似度的类实现

第3章讨论过 Solr 中相关度的 DefaultSimilarity 类的实现。一直以来，tf-idf 向量空间余弦相似度计算是被硬编码到 Lucene 与 Solr 的相关度处理中的。

近几年涌现出许多其他的 Similarity 类实现，Solr 现在已经明确支持 Similarity 类的其他实现，既可以基于 schema，也可以基于字段来实现。这些相似度的类实现不局限于传统的相关度统计处理，如 tf、idf 与规范化，它们还会在 Solr 索引中存储各自的相关度统计信息，以备后用。

要更改用于文档评分的 Similarity 类，可以在 schema.xml 中修改全局相似度类或在字段上自定义一个相似度类。要对相似度进行全局修改，需要在 schema.xml 的主要配置部分添加如下内容：

```
<similarity class="solr.DFRSimilarityFactory">
  <str name="basicModel">I(F)</str>
  <str name="afterEffect">B</str>
  <str name="normalization">H2</str>
</similarity>
```

这个例子重新定义了全局相似度类，从 DefaultSimilarityFactory 变为 DFRSimilarityFactory 类（如表 16.1 所示），该类的必要输入参数（basicModel、afterEffect 与 normalization）是在 XML 区块中进行指定的。如要创建无参数的 Similarity 类，则代码如下所示：

```
<similarity class="my.package.MyCustomSimilarity" />
```

如果不需要全局性重定义 Similarity 类（实际上，开箱即用的 DefaultSimilarity 类适用于大多数情况），你也可以针对特定字段重新定义 Similarity 类，具体做法是在这些字段的配置中定义 Similarity：

```
<fieldType name="text_custom_similarity" class="solr.TextField">
  <analyzer>
    ...
  </analyzer>
  <similarity class="my.package.MyCustomSimilarity" />
</fieldType>
```

表 16.1 Solr 中可用的 Similarity 实现

相似度的类	说明
DefaultSimilarity	Solr 默认相似度的类在第 3 章 (3.2 节) 中已经详细解释过。基于 tf-idf 及几个规范化因子, 使用向量空间余弦相似度模型
BM25Similarity	另一种基于概率模型的 tf-idf 相似度实现。学术研究指出, 在许多文档集合上, BM25 算法的性能要优于 DefaultSimilarity
DFRSimilarity	DFR 随机发散评分公式会考虑每个文档中词频的相对分布 (相对于其他文档), 以确定该文档的词语的相对重要性
IBSimilarity	基于信息的 IB 评分公示是一个较新的算法。它与 DFRSimilarity 类似, 但输入更简单
LMDirichletSimilarity	使用语言模型, 对语料库中的词项权重进行贝叶斯平滑
LMJelinekMercerSimilarity	也使用语言模型, 但评分实现比 LMDirichletSimilarity 更简单
SweetSpotSimilarityFactory	DefaultSimilarity 类的扩展, 为确定“最佳位置”(sweet spot) 的最优 tf 与 lengthNorm 统计提供调整选项。允许设置合理的限制。若超过此限制, 额外的词项无法继续增加 (对于 tf 而言) 或减少 (对 lengthNorm 而言) 相似度得分

虽然可以根据需求自定义 Similarity 类, 但这属于高级操作, 可能会对搜索应用的品质造成显著影响。Solr 提供了开箱即用的 Similarity 替代类, 它们源于信息检索专业研究。因此在全文搜索中, 如果 DefaultSimilarity 类无法满足需求, 可以考虑尝试其他算法实现。我们无法对每一种 Similarity 类的技术细节一一展开, 表 16.1 概述了各种可用算法。

如果想对 Solr 中各种核心相关度算法优劣进行评测, 有多种选择。表 16.1 中大多数相似度实现包含了长长的输入参数, 需要花时间去理解 Solr 的代码, 并且可能还要阅读有关算法的研究论文。虽然你可能乐于深入的学术研究, 但确实存在一些较为简单的、不那么理论化的相关改进方法。在查询阶段提供更多的信息, 包括领域知识、用户行为及用户偏好, 有助于快速、有效地改进搜索应用的相关度。本章其余小节会介绍如何集成此类信息来改进相关度, 以及对搜索相关度改进的总体影响进行评估。

## 16.5 个性化搜索与推荐

随着开源技术 (如 Solr) 的发展, 搜索技术已经商品化, 企业的投入从基础的关键词匹配转向解决搜索应用更困难的问题, 其中一个问题是如何超越事务型搜索体验, 即用户需要输入关键词, 浏览搜索结果, 通过分面过滤出特定文档, 运用高亮技术提供内容的初步查看结果。企业已着手增强传统的搜索体验, 为用户个人信息建档, 根据位置、类别以及已知的用户的其他兴趣点, 提供更加个性化的搜索结果。



与此同时，许多企业（特别是网络公司）还发现，要保持用户的参与度，通常需要给用户推送信息，而不是要求用户直接与搜索应用进行交互。这完全颠覆了现有的搜索范式，搜索系统需要足够智能化，向用户推荐信息，而不是让用户自己去搜索。众所周知，像 Netflix 与 Amazon 这样的企业以推荐系统闻名，研发经费高达数百万美元。不过，以 Solr 为基础也有可能较为轻松地开发此类推荐系统，以显著改善搜索应用的相关性。

### 16.5.1 搜索 vs. 推荐

如果要简单描述一下搜索引擎是什么，我们脑海里浮现的典型画面一般是一个关键词搜索框，某些情况下是一个单独的定位框。同样地，如果要简单描述一下推荐引擎是什么，我们脑海里浮现的一般是神奇的算法，它会根据过去的行为和偏好自动地给出建议。事实上，搜索与推荐都是匹配的表现而已。搜索引擎通常对查询与文档中的关键词与位置进行匹配，推荐引擎则是根据用户之间表现出来的类似行为来推荐文档，或者对两个文档的内容进行匹配。

从基本层面上看，搜索引擎与推荐引擎的工作原理相同：通过建立文档之间链接的稀疏矩阵，使用某种相似性度量来寻找最佳匹配。这些矩阵包括“词项—文档”矩阵（关键词搜索中典型的倒排索引）和“偏好—文档”矩阵（用于推荐的行为的倒排索引，称为协同过滤）。

那么，搜索与推荐的真正区别在于：搜索一般是需要用户输入的手工任务，而推荐通常是了解用户的有关情况，自动地提供建议信息。一种观点认为，推荐是一种自动化搜索，非人工判断哪些是用户想要的相关内容。

与其将 Solr 视为一个文本搜索引擎，不如将其视为一个匹配引擎，它能对已解析的文本进行匹配。搜索是人工的还是自动化的，这与 Solr 没有直接关系。事实上，一些企业已经成功地运用这种思路，直接在 Solr 之上构建推荐系统。后面的小节会介绍如何构建 Solr 驱动的推荐引擎，最终将用户驱动搜索体验理念与自动化的推荐系统相融合，提供强大的个性化搜索体验。

具体来说，我们会讨论一些基于内容的推荐方法，包括基于属性的匹配、基于层级分类的匹配，基于抽取的兴趣物品的匹配（更多类似结果）、基于概念的匹配以及地理位置的匹配。我们还会讨论协同过滤技术，基于用户与内容的交互进行推荐，让 Solr 从用户行为中不断学习，通过返回的更多相关结果反映其智能化程度。最后，我们将这些方法在推荐查询中组合起来，作为更加个性化的搜索体验基础。

### 16.5.2 基于属性的匹配

Solr 能对分本进行分词，生成倒排索引，因此它也能索引用户感兴趣的结构化



文档信息，如类目、位置及其他属性。为构建基于属性的推荐引擎，需要确保用户与文档具备相同类型的描述属性。例如，在求职搜索应用中，对工作职位的描述可能包括地点、薪水与岗位名称。如果用户申请这些职位，就会从用户那里搜集他们的简历、个人资料以及搜索行为等类似的信息。

为方便讲解后续章节内容，这里创建一个求职搜索用例的 Solr 内核。首先，我们向预先配置好的求职搜索 Solr 内核提交一些文档：

```
java -Durl=http://localhost:8983/solr/jobs/update/csv
  -Dtype=text/csv -jar post.jar ch16/documents/jobs.csv
```

求职搜索引擎准备就绪后，接下来考虑一个问题，如何根据用户的个人信息生成一个自动化查询，实现工作岗位的推荐。例如，以下是搜索应用中用户 Jane 的个人资料：

```
Profile:{
  Name: "Jane Doe",
  Industry: "healthcare",
  Locations: "Boston, MA",
  JobTitle: "Nurse Educator",
  Salary:{
    min:40000,
    max:60000
  },
}
```

Jane 自己没有主动去搜索，根据她的个人资料，我们可以轻松地在职位索引上生成一个自动化搜索，为她推荐相关职位。代码清单 16.5 是推荐查询的一个例子。

#### 代码清单 16.5 根据用户个人资料实现基于属性的推荐查询

找到需要  
的职位，对  
精确匹  
配的职  
位赋予  
高权重。  
落在  
Jane 的  
薪金期  
望区间  
的工作  
被赋予  
更高的  
得分。

**查询请求**

```
http://localhost:8983/solr/jobs/select?
  fl=jobtitle,city,state,salary&
  q=(jobtitle:"nurse educator"^25 OR jobtitle:(nurse educator)^10)
  AND ((city:"Boston" AND state:"MA")^15
  OR state:"MA")
  AND _val_:"map(salary, 40000,60000,10,0)"
```

找到 Jane  
所在的精确  
区域的工作.....

....., 同时也包含 Jane  
所在州的工作，但对它们  
赋予相对较低的权重。

**搜索结果**

```
{
  ...
  "response":{"numFound":22,"start":0,"docs":[
    {
      "jobtitle":"Clinical Educator (New England/ Boston)",
      "city":"Boston",
      "state":"MA",
      "salary":41503},
```

波士顿的相关  
工作显示在最  
前面（与期望  
一致）。

```
{
  "jobtitle": "Nurse Educator",
  "city": "Braintree",
  "state": "MA",
  "salary": 56183},
{
  "jobtitle": "Nurse Educator",
  "city": "Brighton",
  "state": "MA",
  "salary": 71359},
```

马萨诸塞州其他城市的相  
关工作排在其次。

薪水在期望区间之外的工  
作仍然返回，但权重相对  
较低。

第一，用户个人资料的许多属性与被搜索的内容可以很好地进行匹配。这适用于大多数搜索应用，但不是全部。第二，搜索中要求同时出现职位名称子句与地点子句，两者都允许进行精确匹配（具有较高的相关度权重）和模糊匹配。这是特定领域的选择，虽然求职者可能对地点与职位名称都很在意，但他们可能没有对查询中这些因素如何进行匹配做出限制。在这种情况下，该查询会匹配同一州内的任何地点，不要求搜索结果在同一短语里同时包含 nurse 和 educator 两个词项。

第三，虽然匹配 Jane 期望的薪金范围的工作被提供了一个 boost 值，但这仅是针对字段提升，而不是过滤器。原因可能在于，不是所有职位都包含薪金范围，或者觉得薪金可能没那么重要吧（换作是你，想要忽略掉那些包更高薪金的职位吗？）第四，查询中没有包含 Jane 个人资料中所写的行业。如果登记了职位的所属行业，这可能会对查询增加额外价值。但演示就止于此，我们没有必要总是如此谨慎地在推荐查询中用到所有信息。

最后，为什么地点子句搜索的是城市和州名，而不使用半径过滤器呢？这样做是为了保持示例足够简单。最终要为查询构造寻找最适合的方式，以及选用什么样的数据来优化搜索应用的相关度。这里更适合的方法可能是使用基于地理距离计算的半径过滤器，具体做法参见第 15 章。使用 Jane 个人资料的举例旨在说明，通过创建自动化搜索来提供推荐的容易程度。

### 16.5.3 分层匹配

在上一节基于属性的匹配讨论中，我们发现，并不是所有属性都是以相同方式创建的，有时需要对具体属性作出选择。如果属性匹配，就要为它们赋予更高权重。这就是分层匹配背后的思想。假设对用户和内容进行分类，将其归入某种层级中（从一般类目到具体类目），这样就可以对这个层级进行查询，对更加专指的匹配赋予更高的相关度权重。

回到前面的例子，现在将 Jane 归入我们认为她可能感兴趣的类目中，为 Jane 个人资料添加所属领域：

```
Janes_Profile:{
  MostLikelyCategory:"healthcare.nursing.oncology",
  2ndMostLikelyCategory:"healthcare.nursing.transplant",
  3rdMostLikelyCategory:"educator.postsecondary.nursing",
  ...
}
```

在 Jane 的个人资料和职位索引中, 假设每个条目包含三级分类, 每一级用句点分开, 就像字段分析器用句点进行分词一样。代码清单 16.6 包含了一个示例查询, 根据层级分类的模糊组合, 返回匹配到的文档。

#### 代码清单 16.6 基于层级分类的匹配

```
http://localhost:8983/solr/jobs/select?
df=classification&
q=(
  (
    "healthcare.nursing.oncology"^40
    OR
    "healthcare.nursing"^20
    OR
    "healthcare"^10
  )
  OR
  (
    "healthcare.nursing.transplant"^20
    OR
    "healthcare.nursing"^10
    OR
    "healthcare"^5
  )
  OR
  (
    "education.postsecondary.nursing"^10
    OR
    "education.postsecondary"^5
    OR
    "education"
  )
)
```

这是与 Jane 简历  
“最为接近”(第一  
级)的目录层级。

与 Jane 简历 “其  
次接近”(第二级)  
的目录层级。

与 Jane 简历 “较  
为接近”(第三级)  
的目录层级。

你可能会注意到该查询的结构特点。首先, 该查询将 Jane 个人资料的每个类目分成三个词项, 分别对应一个分类层级 (healthcare.nursing.oncology vs. healthcare.nursing vs. healthcare)。其次, 每个词项被赋予不同的查询权重, 词项越专指, 其权重越高。这样做是为了在搜索结果中对更具体的(同时可能是更好的)匹配文档进行提升。再次, 三个查询集合分别对应 Jane 个人资料的三个潜在类目 (healthcare.nursing.oncology、healthcare.nursing.transplant 与 educator.postsecondary.nursing)。

“最有可能”的类目或许是不正确的，或者所有类目都添加了一些值，这也是可能的。在查询中也包含替代类目，但其权重较低。最终结果是，使用词项的查询权重将它们的概率（从最可能到最不可能）与专指度（从泛指到专指）结合起来，通过构造模糊查询来匹配满足以上条件的文档，同时对搜索结果顶部的文档进行提升，这些文档是匹配到层级中最佳属性组合的文档。

## 16.5.4 更多类似结果

迄今为止，基于属性和层级的推荐方法都是基于用户的结构化信息，根据要搜索的内容生成自动化查询。除此之外，还有一些方法可以使用非结构化内容来构建自动化推荐。如果用户对某个特定文档感兴趣，或者反过来说，文档描述了用户的兴趣（例如，用户个人资料或简历），那么就没有必要根据该文档中抽取的结构化信息进行推荐。

相对于用户输入关键词，为实现基于文档内容的推荐，开箱即用的 Solr 配备了一个可配置的请求处理器和一个搜索组件。Solr 的更多类似结果处理器能够处理任何文档（通过索引过的文档 ID 或传递的文档文本），它从文档中抽取感兴趣的词项，使用这些词项进行关键词搜索，以此寻找类似的文档。从内部看，更多类似结果处理器会从文档中抽取感兴趣的词项，将文档视为词项向量，根据 tf-idf 相似度计算，抽取匹配度最高的词项。然后，将这些排名高的词项组成一个查询，用来查找其他类似的文档。

为了使用更多类似结果处理器，需要在 solrconfig.xml 中启用它：

```
<requestHandler name="/mlt" class="solr.MoreLikeThisHandler" />
```

启用了更多类似结果处理器，就可以查找相似的文档了。出于演示目的，我们在 16.5.2 节的求职搜索内核上提交以下查询：

```
http://localhost:8983/solr/jobs/mlt/?  
  df=jobdescription&  
  q=J2EE&  
  mlt.fl=jobtitle,jobdescription
```

该查询将查找包含关键词 J2EE 的文档，找到最匹配的相关文档，然后对 jobtitle 和 jobdescription 字段进行统计分析。值得注意的是，更多类似结果这个功能需要在 schema.xml 中对统计分析字段设置 termVectors="true" 或 stored="true"（通过 mlt.fl 参数指定）。启用词项向量会快一些，这是因为更多类似结果的实现需要在查询阶段处理已存储的内容来获取词项向量。代码清单 16.7 给出了该查询返回的示例推荐，通过设置额外的参数 mlt.interestingTerms=details，返回用于推荐查询的词项的有关信息。

## 代码清单 16.7 "J2EE" 查询返回的更多类似结果

## 查询请求

```
http://localhost:8983/solr/jobs/mlt?
df=jobdescription&
fl=id,jobtitle&
rows=5&
q=J2EE&
mlt.fl=jobtitle,jobdescription&
mlt.interestingTerms=details
```

## 搜索结果

```
{
  "match": {"numFound": 122, "start": 0, "docs": [
    {
      "id": "fc57931d42a7ccce3552c04f3db40af8dabc99dc",
      "jobtitle": "Senior Java / J2EE Developer"
    }
  ]},
  "response": {"numFound": 2225, "start": 0, "docs": [
    {
      "id": "0e953179408d710679e5ddbd15ab0dfae52ffa6c",
      "jobtitle": "Sr Core Java Developer"
    },
    {
      "id": "5ce796c758ee30ed1b3da1fc52b0595c023de2db",
      "jobtitle": "Applications Developer"
    },
    {
      "id": "1e46dd6be1750fc50c18578b7791ad2378b90bdd",
      "jobtitle": "Java Architect/ Lead Java Developer -
        WJAV Java - Java in Pittsburgh PA"
    },
    {
      "id": "4735d1f62503c330c74e470d2d0a26fa855a4257",
      "jobtitle": "Java Developer / Software Developer ( 5 - 7 + Openings)"
    },
    {
      "id": "2d27ee49cba8189035367f0e048bae07e0faae30",
      "jobtitle": "Java Developer"
    }
  ]},
  "interestingTerms": [
    "jobdescription:j2ee", 1.0,
    "jobdescription:java", 1.0,
    "jobdescription:senior", 1.0,
    "jobtitle:developer", 1.0,
    "jobdescription:source", 1.0,
    "jobdescription:code", 1.0,
    "jobdescription:is", 1.0,
    "jobdescription:client", 1.0,
    "jobdescription:our", 1.0,
    "jobdescription:for", 1.0,
    "jobdescription:a", 1.0,
    "jobdescription:to", 1.0,
    "jobdescription:and", 1.0
  ]
}
```

1 初始的 J2EE 查询。

2 关注词项的字段设定。

3 请求返回兴趣词项及相关文档。

与查询匹配的最佳文档，用于寻找兴趣词项。

推荐文档列表。

4 推荐列表的第一个文档包含兴趣词项。

我们对代码清单 16.7 中更多类似结果的请求进行详细说明。第一步，指定一

个查询 ❶，找到排名最靠前的搜索结果，作为推荐的依据。在很多情况下，你可能对推荐什么样的文档已经心里有数，它们最好是查询返回的最佳匹配文档，例如，`q=id:fc57931d42a7ccce3552c04f3db40af8dabc99dc`。请求中还应指明 ❷在哪些字段上查找用户可能感兴趣的词项。更多类似结果处理器会对最佳查询匹配文档中的每个词项进行检查，通过比较每个词项的 `tf-idf` 得分来决定潜在的显著性，之后最感兴趣的词项通过一个较长的 `OR` 查询返回 ❸，这样就得到了一个推荐文档列表。

如果想深入了解该查询中用到的词项，可以在查询中添加 `mlt.interestingTerms=details` 参数 ❹，返回那些用于推荐查询的兴趣词项列表。代码清单 16.7 显示每个词项的得分为 1.0，这表明它们在推荐查询中的权重相同，不过我们仍然可以设置 `mlt.boost=true` 参数，通过 `tf-idf` 计算来表示每个词项的 `boost` 值。

### 代码清单 16.8 更多类似结果请求中提升的兴趣词项

#### 查询请求

```
http://localhost:8983/solr/jobs/mlt?
df=jobdescription&
q=id:fc57931d42a7ccce3552c04f3db40af8dabc99dc &
mlt.fl=jobtitle,jobdescription&
mlt.interestingTerms=details&
mlt.boost=true
```

← 根据词项的 `tf-idf` 得分，调整兴趣词项的权重。

#### 搜索结果

```
{
  ...
  "response": {"numFound": 301764, "start": 0, "docs": [
    {
      "id": "0e953179408d710679e5ddbd15ab0dfae52ffa6c",
      "jobtitle": "Sr Core Java Developer"},
    {
      "id": "1e46dd6be1750fc50c18578b7791ad2378b90bdd",
      "jobtitle": "Java Architect/ Lead Java Developer -
        WJAV Java - Java in Pittsburgh PA"},
    ...
  ]},
  "interestingTerms": [
    "jobdescription:j2ee", 1.0,
    "jobdescription:java", 0.68131137,
    "jobdescription:senior", 0.52161527,
    "jobtitle:developer", 0.44706684,
    "jobdescription:source", 0.2417754,
    "jobdescription:code", 0.17976432,
    "jobdescription:is", 0.17765637,
    "jobdescription:client", 0.17331646,
    "jobdescription:our", 0.11985878,
    "jobdescription:for", 0.07928475.
```

❶ 请注意，兴趣词项权重调整后，结果排序发生了变化。

← 返回每个词项的 `tf-idf` 权重值。  
❷

```
"jobdescription:a",0.07875194,
"jobdescription:to",0.07741922,
"jobdescription:and",0.07479082]}}
```

通过获取文档的兴趣词项以及它们的 tf-idf 权重，我们甚至可以在搜索应用层中使用这些词项。你可能会注意到，推荐结果看起来与代码清单 16.7 略有不同。其中，第二个结果 ❶ 是有关“Java Architect...”职位，而不是“Applications Developer”职位。这是因为推荐查询中用到的每个兴趣词项都被根据 tf-idf 计算进行了提升，所以诸如 j2ee 和 java 的权重变大，而其他每个词项的权重还是 1.0。在更多类似结果处理器得到的 Solr 响应的 interestingTerms 部分可以查看每个词项的相对权重 ❷。

### 外部文档的更多类似结果

迄今为止，查找最佳文档的查询方法也适用于搜索引擎对已有文档进行相关推荐。但是，如果想要匹配的文档不在搜索索引中，该如何处理？所幸，更多类似结果处理器还可以根据传入外部文档的全文进行推荐。为了充分利用这个功能，需要向更多类似结果请求处理器提交一个 HTTP POST 请求，将推荐的文档内容放入 POST 的正文 (body) 部分。如果文档内容不多的话，也可以使用 HTTP GET 请求，通过 stream.body 参数向 Solr 传入内容。代码清单 16.9 介绍了这种文档传入方式。另外，你也可以使用其他请求处理器传入文档内容，例如，/update 处理器。

#### 代码清单 16.9 为外部文档推荐更多类似结果

##### 查询请求

```
http://localhost:8983/solr/jobs/mlt?
df=jobdescription&
mlt.fl=jobtitle,jobdescription&
mlt.interestingTerms=details&
mlt.boost=true&
stream.body=Solr is an open source enterprise search platform from the
Apache Lucene project. Its major features include full-text search, hit
highlighting, faceted search, dynamic clustering, database integration,
and rich document (e.g., Word, PDF) handling. Providing distributed
search and index replication, Solr is highly scalable. Solr is the most
popular enterprise search engine. Solr 4 adds NoSQL features.
```

##### 搜索结果

```
{
  "response":{"numFound":2211,"start":0,"docs":[
    {
      "id":"eff5ac098d056a7ea6b1306986c3ae511f2d0d89",
      "jobtitle":"Enterprise Search Architect - Plymouth, MN; Hartford,
        CT; Cypress, CA; Salt Lake City, UT or Telecommute"},
    {
      "id":"37abb52b6fe63d601e5457641d2cf5ae83fdc799",
      "jobtitle":"Sr. Java Developer"},
    ...
  ]}
```

待匹配的外部文档的  
文本内容。



```
{
  "id": "349091293478dfd3319472e920cf65657276bda4",
  "jobtitle": "Java Lucene Software Engineer"},
{
  "id": "8139a347d87ab24ffe6895de800d31d338b216ea",
  "jobtitle": "Websphere Java Lead"},
{
  "id": "f10a09ac5a30ec7743c67198a3cec50a0aafdef6",
  "jobtitle": "Sr. Java Engineer with Search / Algorithm experience"}]
},
"interestingTerms": [
  "jobdescription:search", 1.0,
  "jobdescription:solr", 0.9155779,
  "jobdescription:features", 0.36472517,
  "jobdescription:enterprise", 0.30173126,
  "jobdescription:is", 0.17626463,
  "jobdescription:the", 0.102924034,
  "jobdescription:and", 0.098939896]]
```

任何文档都可以传入更多类似结果处理器进行分析。更多类似结果处理器会根据字段指定的 `mlt.fl` 参数对传入的文本进行分析。之后，已分析的文本会被视为 Solr 索引中的一个文档。

### 更多类似结果的改进可能

虽然更多类似结果处理器非常灵活，但需要注意两点。首先，你可能已经注意到，兴趣词项列表中许多词项都是对源文档的描述，例如，代码清单 16.9 的 `solr` 与 `search`；其次还存在一些噪声词项，例如，`and`、`is` 与 `the`。通常使用停用词表可以解决这些噪声词项。另一种办法是构建词性分析器，仅使用文本中的名词。虽然“好的”词项往往比“坏的”词项权重高，但推荐结果仍有可能包含无用的词项，所能做的就是降低噪声来改善搜索结果的总体质量。

本节介绍了更多类似结果处理器的主要功能，还有许多其他选项可用于改善搜索结果。另外，除了将更多类似结果作为单独的请求处理器使用，我们还可以将它作为典型搜索查询的一个搜索组件。有关这些扩展功能的更多信息，请访问 Solr 更多类似结果功能的 Wiki 页面 <http://wiki.apache.org/solr/MoreLikeThis>。

我们已经了解了 Solr 如何从源文档中提取兴趣词项，基于的是大量文本进行文档推荐的方法。不过，如果没有大量的文本（完整的文档），又该如何描述用户的兴趣呢？下一节将讨论如何从最少的信息，甚至只有一两个关键词，根据找到能实现有用推荐的相关概念。

## 16.5.5 基于概念的匹配

一般 Solr 都是根据用户输入的关键词来搜索相关文档。前面两小节介绍了基于属性（如类目）或者整个文档的搜索方法。在这两种情况下，搜索用到的词项直接来自于查询，即使是更多类似结果处理器，兴趣词项也是源自查询所得的文档的。

太多的限制对推荐来说不是什么好事。如果能找到相关但不同的内容，这会很有价值。本节介绍 Solr 的聚类组件，用于发现文档之间的相似度，找出初始查询或文档没有通过字面表达出来的相关概念。

聚类组件是一个搜索组件，使用 SearchHandler 可以将其添加到任意请求中。它会在一个查询的搜索结果中寻找相似的词项或短语。为了解释搜索结果聚类的基本原理，我们以公共求职搜索引擎中查询 .Net Jobs 的搜索结果为例，如图 16.1 所示。请注意，图 16.1 没有使用本书示例章节的数据，如果搜索之前小节中的求职索引，得到的搜索结果与之类似。

初始搜索: .Net Jobs

---

C# Developer - .Net Programmer - Software Engineer - C#  
**CyberCoders Engineering**  
FL - Boca Raton (posted 3 Weeks Ago)  
This position is open as of 2/12/2012.C# Developer - .Net Programmer - Software Engineer - Senior C# Developer. If you are a Senior Software Engineer or...

---

Senior .NET Developer-C# & C++, Client Server, Multithreading  
**WSI Nationwide**  
NY - Manhattan (posted 3 Weeks Ago)  
Qualified Senior .NET Developer candidates will have 5+ years designing and building successful Windows products with strong OOD/OOP skills, and...

---

Sr .Net / Lead developer  
**GDI Infotech**  
MI - Detroit (posted 1 Week Ago)  
Title: Sr .Net / Lead developer Location: Flint, MI / Detroit, MI Duration : 6 Months Contract or Contract To Hire or Full Time Highly educated...

---

.Net Software Engineer / .Net Developer  
**AMS Staffing Solutions, L.L.C.**  
SD - Rapid City (posted 2 Weeks Ago)  
Please send resume in Word format if you are interested in this .Net Software Engineer opening in Rapid City, SD. Client is looking to pay between \$...

图 16.1 在典型的求职搜索引擎中查询 .Net Jobs 的搜索结果

如果初始查询是一个具体的关键词，如图 16.1 所示，那么查询结果很可能不仅与该关键词本身相关，而且与该关键词相似的概念相关。如果查看 .Net Jobs 的搜索结果，我们会发现与搜索相关的其他关键词和短语，例如，software engineer（软件工程师）、developer（开发者）、c# 以及 .net developer（.net 开发者）等，多次出现在了搜索结果的文本中。图 16.2 高亮显示了这些相关的词项和短语。

初始搜索: .Net Jobs

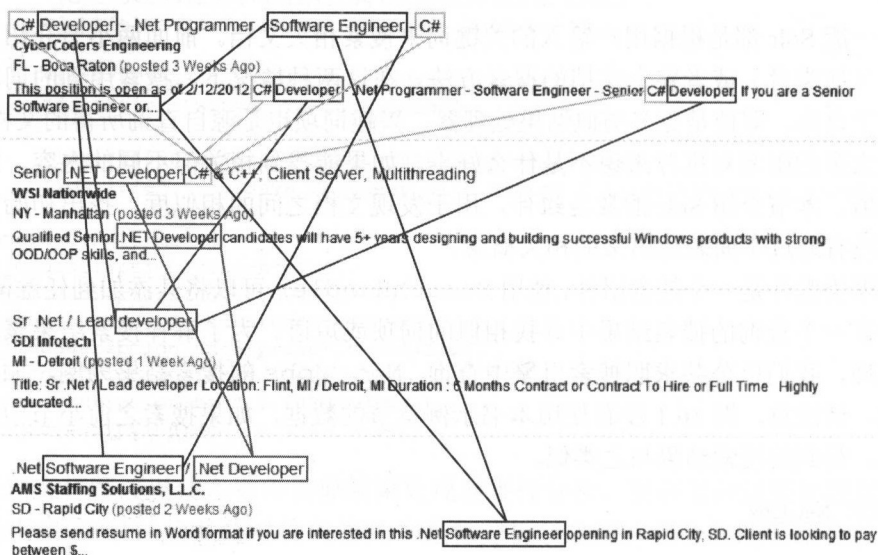


图 16.2 在关键词查询的搜索结果中对相似单词或词组进行聚类，从而发现相关概念

正如图 16.2 所示，对顶部返回的文档（根据相关度得分排序）进行关键词聚类，这样可以推断出与原始查询相关的其他概念。使用这种方法，我们可以将查询 `q=.Net Jobs` 扩展如下：

```
q=.Net Jobs OR ("software engineer" OR "c#"
               OR ".net developer" OR "developer")^0.25
```

此查询有效地扩展了搜索结果，包含了其他相关概念，并对这些概念赋予低于原始词项的权重。如果已经返回太多结果，想要提升 `.Net Jobs` 原始查询概念相关的结果，则可以对查询做如下修改：

```
q=.Net Jobs AND (*:* OR "software engineer" OR "c#"
                 OR ".net developer" OR "developer")
```

该查询不会改变原始查询返回的结果数，但是会对搜索结果重新排序，以使得相关概念的匹配文档在搜索结果中排名靠前。如果得到太多的搜索结果，其中许多结果的匹配质量不佳，那是因为它们仅匹配了特定关键词，但概念上并不相关。将这些概念作为查询的一部分，这样能有效消除那些既不匹配原始查询也不包含相关概念的文档：

```
q=.Net Jobs AND ("software engineer" OR "c#"
                 OR ".net developer" OR "developer")
```

根据匹配系统的具体需求，通过限制查询来改进搜索结果的查准率（通过请求更高水平的概念相关的词项）或查全率（通过取回概念相关但不是专门匹配的文档）的方法有多种。所幸，在 Solr 中可以很容易地启用聚类组件，从搜索结果中得到标记好的聚类，从而用于概念匹配。

### 启用搜索结果聚类

要启用 Solr 的搜索结果聚类，需要在 `solrconfig.xml` 中将 `ClusteringComponent` 类添加到搜索处理器的搜索组件列表。代码清单 16.10 给出了启用聚类的 `SearchHandler` 创建方法。

代码清单 16.10 在 `solrconfig.xml` 中启用聚类组件

```
<searchComponent name="clustering" enable="true"
  class="solr.clustering.ClusteringComponent">
  <lst name="engine">
    <str name="name">default</str>
    <str name="carrot.algorithm">
      org.carrot2.clustering.lingo.LingoClusteringAlgorithm
    </str>
    <str name="carrot.resourcesDir">clustering/carrot2</str>
  </lst>
</searchComponent>

<requestHandler name="/clustering" enable="true"
  class="solr.SearchHandler">
  <lst name="defaults">
    <bool name="clustering">true</bool>
    <str name="clustering.engine">default</str>
    <bool name="clustering.results">true</bool>
    <str name="fl">*,score</str>
  </lst>
  <arr name="last-components">
    <str>clustering</str>
  </arr>
</requestHandler>
```

步骤 1：定义聚类搜索组件。

定义一个聚类“引擎”（算法及配置）。

Lingo 是默认的 Carrot2 聚类算法，适用于许多数据集。

使用之前定义的聚类“引擎”。

步骤 2：在 `SearchHandler` 定义中启用聚类搜索组件。

当定义好聚类组件并添加到搜索处理器之后，就可以在任何搜索请求中检索已标记的聚类。代码清单 16.10 中定义了一个新的请求处理程序（`/clustering`），它专门用于聚类搜索，如果愿意，也可以把它添加到主请求处理器（`/select`）中。为了检索聚类后的搜索结果，我们需要提交一条查询，请求已聚类的搜索结果：

```
http://localhost:8983/solr/jobs/clustering?
```

```
q=content:(solr OR lucene)&
```

```
rows=100&
```

```
carrot.title=jobtitle&
```

```
carrot.snippet=jobtitle&
```

```
LingoClusteringAlgorithm.desiredClusterCountBase=25
```

该请求指定了几个参数，如果愿意，也可以在请求处理器的默认配置中指定这些参数。rows 参数表示返回多少个搜索结果，这与聚类组件的文档数相同，将用于聚类文本。该聚类组件使用标题字段（通过 carrot.title 参数指定）和片段字段（通过 carrot.snippet 参数指定）。通常，文档的标题通常较短，更能代表文档，而内容片段对聚类操作也很重要。在本例中，我们将标题和片段都指向同一字段。

LingoClusteringAlgorithm.desiredClusterCountBase 参数定义了要返回的聚类的理想数量。这不能保证找到确切数量的聚类，但为算法提供了目标，它可以从一般（较少聚类）到具体（多个聚类）区间中选择所需的聚类。聚类算法有好几种，每种都需要具体配置。关于聚类的更多信息，请访问 Solr 的 wiki 页面 <http://wiki.apache.org/solr/ClusteringComponent>。代码清单 16.11 是查询的搜索结果。

### 代码清单 16.11 聚类文档

#### 查询请求

```
http://localhost:8983/solr/jobs/clustering?
```

```
q=content:(solr OR lucene)&
```

```
fl=id,jobtitle&
```

```
rows=100&
```

```
carrot.title=jobtitle&
```

```
carrot.snippet=jobtitle&
```

```
LingoClusteringAlgorithm.desiredClusterCountBase=25
```

#### 搜索结果

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 46,
    "response": { "numFound": 62, "start": 0, "docs": [
      {
        "id": "349091293478dfd3319472e920cf65657276bda4",
        "jobtitle": "Java Lucene Software Engineer",
        {
          "id": "bea2d65c7786f6fccealc134f1f743f20b10a1",
          "jobtitle": "Java Developer",
          ...
        ] },
        "clusters": [ {
          "labels": ["Software Engineer"],
          "score": 15.47007225959035,
          "docs": [ "349091293478dfd3319472e920cf65657276bda4",
            "c6e08b25102353da348334c8cbd0018fa8912559",
            "76ab5e51d759a93613d056d1930845a211aabee6",
            "b5bdbf64ce704b927992060e5eb02fcef99a0cf",
```

原始文档必须包含用于聚类的字段。

类簇包含文档的 ID。

类簇来自 carrot.title 和 carrot.snippet 字段。

```

...
    "c643b258bc1fae773b3059abe0b3acc0befd5f34"]},
{
  "labels":["Java Developer"],
  "score":12.731272597053874,
  "docs":[...]},
...
{
  "labels":["Software Developer"], ...
},
...
{
  "labels":["Systems Engineer"], ...
},
{
  "labels":["Web Developer"], ...
{
  "labels":["Search"], ...
{
  "labels":["Senior Java Developer"], ...
}
{
  "labels":["Data Architect"], ...
}
{
  "labels":["Data Developer"], ...
}
...
{
  "labels":["Other Topics"],
  "score":0.0,
  "other-topics":true,
  "docs":["34fe10d760d2d7b34588acc0c25b90599859ed84",
    "7fb1c9039228cb9649b0c403cd4bbb2a321a0c98",
    "0b5dcc61b81cbf6d3773a5d45d746d0b3b55d587",
    "e921ee1f92fb317f8d11877b81ee71548b153341",
    ...
    "c6c92acd461d997ae9471cf46f30698f93d198d7"]]}]}

```

类簇来自 carrot.  
title 和 carrot.  
snippet 字段。

类簇来自 carrot.  
title 和 carrot.  
snippet 字段。

无法形成聚类的文档  
归入 “other-topics”。

你会发现，聚类组件试图根据与查询匹配的最佳文档的文本来寻找相似的聚类。聚类响应包括聚类文档 ID 列表以及一个聚类标签。这个标签代表该聚类中文档共同拥有的最具描述性的词项或短语。聚类标签可作为显示给用户的一种动态生成的分面，或者将聚类标签作为相似概念对原始查询进行扩展。

将聚类标签作为相关概念，可以将用户的搜索扩展到其他相似概念。将代码清单 16.11 的聚类标签转换成关键词查询，这样会得到相似的搜索结果，如代码清单 16.12 所示。

## 代码清单 16.12 基于相关 / 聚类概念的推荐

## 查询请求

```
http://localhost:8983/solr/jobs/select?
df=jobdescription&
fl=id,jobtitle&
q=(solr OR lucene) OR "Java Engineer" OR "Software Developer"
```

## 搜索结果

```
{
  ...
  "response":{"numFound":196,"start":0,"docs":[
    {
      "id":"f10a09ac5a30ec7743c67198a3cec50a0aafdef6",
      "jobtitle":"Sr. Java Engineer with Search / Algorithm
        experience"},
    {
      "id":"1612550afd8e536c7db7e75b90cbb5fe29a5ed48",
      "jobtitle":"Senior Java Engineer"},
    {
      "id":"349091293478dfd3319472e920cf65657276bda4",
      "jobtitle":"Java Lucene Software Engineer"},
    ...
  ]}
}
```

少数最佳类簇添加到查询中，用于查询扩展。

相关的聚类添加到查询后返回更多的结果。

如你所见，我们对代码清单 16.11 得到的聚类标签进行搜索，根据原始查询(solr OR lucene)中发现的概念返回了许多类似的文档。这种方式最适用的情况是，当数据量有限时（例如只有用户之前的关键词搜索），希望根据原始关键词搜索自动化建议概念相似的内容。

至此，你已经了解了一些基于内容来扩充搜索结果的方法，包括基于属性和层级分类的搜索，更多类似结果功能（将文档作为查询来推荐与其相似的文档），以及 Solr 的聚类组件（找出与查询相关的概念）。用户考虑推荐的文档是否相关时，文档的内容并不是唯一重要的方面。下一节讨论地理位置邻近度在相关度优化中所发挥的作用。

### 16.5.6 地理位置的匹配

当基于内容进行推荐时，需要考虑用户对所处地理位置的敏感性，这一点很重要。对于一些搜索应用而言，例如餐馆指南、互联网求职公告板以及在线交友网站，如果向用户自动推荐内容，掌握他们的位置（或者他们感兴趣的地点）是非常关键的。如果客户想要休闲郊游，搜索应用却推荐了一个远在千里之外的地方，他们会觉得这个搜索应用不可靠。

反过来，如果是电子商务网站、音乐与电影流媒体网站或是网上书店，客户对



地理位置就不那么在意了，这是因为这类商品可以很容易地快递或传输给他们。

如果用户在意位置，你就应该谨慎地考虑是否要为高度敏感的用户设置严格的位置过滤器（参见 15.2 节），或者为地理位置接近的文档略微提升相关度（参见 16.3.4 节）。地理位置匹配与搜索应用所属的领域密切相关，正确决策的出发点是为用户提供最相关的结果，这一点要把握住。

迄今为止，讨论的推荐相似文档的方法都是基于文档的内容信息的。在下一节中，我们将探讨如何借助用户行为——让用户与文档进行交互，基于群体智慧创建自动化的反馈机制——这样可以极大改进搜索系统的相关度。

### 16.5.7 协同过滤

协同过滤是机器学习领域中最常用的，也是性能最佳的推荐算法之一。与之前介绍的那些推荐方法不同的是，协同过滤不是基于内容相似度，而是基于用户与文档的交互行为。

你可能已经在多处见到协同过滤的运用场景了。在电子商务网站购物如 [www.amazon.com](http://www.amazon.com)，你可能见到过所谓的商品推荐，即购买该商品的用户也会购买其他一些商品。简而言之，协同过滤假设相似的用户会以相似的行为方式与相关内容进行交互。如果有人购买了电视节目生活大爆炸第一季的 DVD，那么他们就有可能购买第二季的 DVD，这是一种合理的推断。如果有足够多的用户同时购买了第一季和第二季的 DVD，那么这种关联就形成了，可以启用协同过滤算法为已经购买（或打算购买）生活大爆炸第一季的用户推荐第二季。图 16.3 展示了协同过滤结果。

事实上，许多用户一次性购买了生活大爆炸的第一季、第二季和第三季的 DVD，那么第二季和第三季可能会被推荐给购买了第一季的用户。在这种情况下，一个好的内容推荐系统可能也会推荐类似的内容，这是因为同类型商品具有相同的特征与关键词。协同过滤还有更有趣的功能，比如，推荐与用户正在考虑的商品不同但相关的商品。图 16.4 给出了一个推荐场景。

在图 16.4 中，用户打算购买一台蓝光播放器，推荐算法为他推荐了诸如 HDMI 线和设备保修计划等不同的商品。这些推荐类型出现在结账页面上是不错的，用户不想购买两台蓝光播放器，但他们可能需要购买一些配件。这个例子显示了协同过滤算法能够对基于内容的推荐算法进行优化：对用户而言推荐的这些文档，内容乍看不是好的匹配结果，却很可能是用户感兴趣的内容。与之类似，基于协同过滤的推荐可能会给购买了婴儿车的用户推荐奶嘴和尿布，给购买了验孕棒的用户推荐 M&M 巧克力豆和其他零食。这些推荐决策的对错与否，都是基于商品之间成组的聚集关联性。

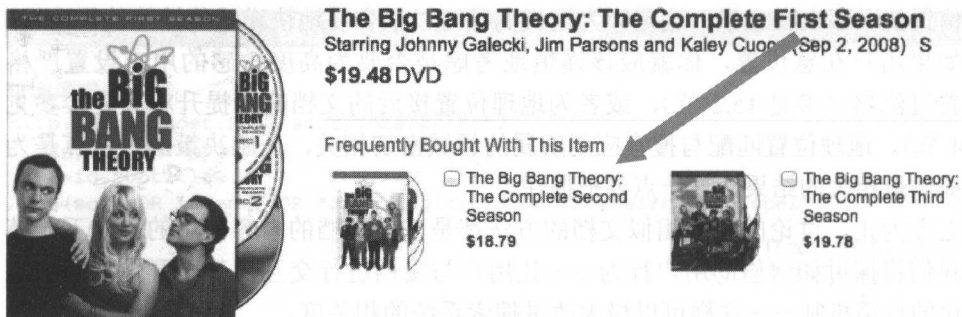


图 16.3 协同过滤举例。多个用户一次性购买某些商品的行为可以说明这些商品有一定关联，未来也会有其他用户需要一次性购买这些商品，这是形成推荐的依据



图 16.4 协同过滤擅长推荐补充型商品，它们与原始商品相关，但不属于同一类型

协同过滤借助集体智慧，根据用户行为对算法进行调整。在实践中，该算法将文档的相似度排名外包给用户，通过用户行为来对基于物品的相关度权重进行调整。正因为如此，在许多行业实践中协同过滤算法优于基于内容的推荐。

### 协同过滤的实现

如何在搜索应用中实现协同过滤呢？Solr在索引中构建“词项—文档”稀疏矩阵，以相同的方法也可以构建“行为—文档”稀疏矩阵。第3章曾介绍过，当Solr接收到文档，它会根据指定的字段对内容进行分析，最终形成倒排文档。假设为电子商务应用构建一个基于行为的搜索，我们可以将用户行为（如购买行为）作为已购买商品文档的字段，如图16.5所示。

在电子商务应用中建立用户购买行为与文档之间的映射，这样就可以在Solr索引文档之间有效地创建链接。假设相似的用户购买相似的商品，这意味着，相似的

用户对应的文档很有可能是相关的。为了利用这些关系向新用户推荐商品，我们需要做的是，找到其他的相似用户，向新用户推荐他们购买过的商品。举例来说，当前用户在购物车里添加了两个文档，分别是 doc1 和 doc4，图 16.6 演示了推荐过程的第一步，根据当前用户购物车里的商品，查找购买过这些商品的其他相似用户。

发送给Lucene/Solr的是什么：

Lucene/Solr如何对内容进行索引（概念性）：

文档	"Users who bought this product" field	词项	文档
doc1	user1, user4, user5	user1	doc1, doc5
doc2	user2, user3	user2	doc2
doc3	user4	user3	doc2
doc4	user4, user5	user4	doc1, doc3, doc4, doc5
doc5	user4, user1	user5	doc1, doc4
...	...	...	...

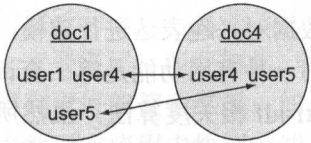
图 16.5 文档的用户偏好映射为文档的字段

正如你所见，根据一个或多个商品寻找相似用户，与根据文档 ID 查找相似用户一样简单。事实上，如果文档与用户的映射以键值方式存储，我们甚至不需要用 Solr 进行这样的查找。你可能想要找到最相似的用户，这需要通过对文档中出现较多的用户赋予较高的权重来实现。在这个例子中，例如，user4 和 user5 同时出现在 doc1 与 doc4，而 user1 仅出现在 doc1 中，所以 user1 相关度低一些。如果要根据用户在文档出现的次数来设置权重，那么需要在 user 字段设置分面（&facet=true&facet.field=user&facet.mincount=1）。无论哪种方式，我们的目标是将排名最靠前的用户映射到想要推荐的文档。一旦有了相似用户列表，下一步就是找到这些相似用户购买过的文档，如图 16.7 所示。

第1步：找到喜欢相同文档的相似用户

q=id:("doc1" OR "doc4")

文档	"Users who bought this product" field
doc1	user1, user4, user5
doc2	user2, user3
doc3	user4
doc4	user4, user5
doc5	user4, user1
...	...



最高得分结果（最相似的用户）：

- 1 user4 （2 个共享喜好）
- 2 user5 （2 个共享喜好）
- 3 user1 （1 个共享喜好）

图 16.6 根据新用户的兴趣词项寻找相似的用户

第2步：搜索相似用户共同喜欢的文档

最相似用户：

- ① user4 (2 个共享喜好)
  - ② user5 (2 个共享喜好)
  - ③ user1 (1 个共享喜好)
- $q=\text{userlikes}:(\text{user4}^2 \text{ OR } \text{user5}^2 \text{ OR } \text{user1}^1)$

词项	文档
user1	doc1, doc5
user2	doc2
user3	doc2
user4	doc1, doc3, doc4, doc5
user5	doc1, doc4
...	...

最值得推荐的文档：

- ① doc1 (匹配 user4, user5, user1)
- ② doc4 (匹配 user4, user5)
- ③ doc5 (匹配 user4, user1)
- ④ doc3 (匹配 user4)

图 16.7 搜索相似用户“喜欢的”文档

协同过滤方法的第二步是，把第一步得到的相似用户转换成一个查询。由于一些用户会出现在多个文档中，因此这些用户会在查询中获得较高的权重。首先，可用的文档越多，能找到的相似用户就越多。更重要的是，有交集的用户会被给予更高的权重，而且如果数量变得太大，为保证查询执行效果，甚至可以潜在地排除一些不重要的用户。如果用户数量很少，那么就应该将他们全部包含在查询中。但是如果不再有其他用户与文档建立关联的话，查询效果会衰减。

执行图 16.7 中的查询，寻找“喜欢”相同文档的用户（图 16.6 已经识别出这些用户），就会得到与查询匹配的搜索结果，根据相关度对它们进行排序。在电子商务示例中，这种“喜欢”表现为购买行为，尽管不是必须要购买。如果搜索应用的用户数量有限或流量有限，在未找到更好的用户兴趣表达机制之前，你可以考虑使用搜索结果点击数来表示“喜欢”。与购买相关的商品相比，获取搜索结果的用户点击数相对容易，但使用点击数会给相关度计算带来额外的噪音，因此需要根据可用的用户行为数据对兴趣表达进行建模，以找到更多相关的结果。

需要注意的另一种有用功能是第二查询(second query)，即搜索相似用户的文档，使用 Solr 默认的 tf-idf 相关度算法。虽然所有文档的 tf 值一般为 1.0（除非对同一用户的文档标记了两次，也可以对该用户设置 boost 值），但 idf 还是会为用户首选的其他文档数量建模。如果搜索应用存在一些垃圾用户（他们对许多不同文档都表达兴趣），使用这种方法会非常有用。根据 idf 的计算，用户感兴趣的文档差异性越大，该用户表达的兴趣越小众。因此，与不加区分地链接许多文档的用户相比，链接较少文档的差异性用户会被认为更有价值。

协同过滤还有一个特点值得一提，它不局限于基于之前的物品发现类似的物品。示例中物品到物品的推荐用来说明协同过滤的概念，但它并不是唯一的实现途径。另外，通过翻转这个模型，我们也可以使用文档链接用户来推荐用户。例如，你可以对用户进行索引，而不对文档进行索引，每个用户的字段显示了该用户喜欢哪些文档。然后，根据某个用户喜欢的文档，其他用户也会被其他用户喜欢的原则，你可以找到与其他用户相似的用户。如果为新商品找到潜在的购买者，有用的做法是基于用户过去的购买行为或其他需求将用户分组。

不论哪种实现方式，协同过滤的优势在于，它不需要了解文档内容。因此，即便只有文档 ID 和用户，你也可以构建 Solr 驱动的推荐引擎。只要有足够的用户与文档进行关联，就可以得到不错的推荐结果。如果不向 Solr 提交任何文本内容、属性或分类，就不能使用这些额外的技术。下一节讨论为什么要组合多种技术，以达到推荐系统的最优相关度。

### 16.5.8 混合方式

本章介绍了多种推荐方法，每种都各有优缺点。现实中往往将这些方法结合起来使用，以改进相关度。由于每一种推荐类型都是在查询中实现的，因此对这些查询的元素进行混合和匹配是一件琐碎的事情。每种方式的优缺点是什么呢？

更多类似结果处理器和聚类组件用于寻找相关概念，擅长处理文档之间基于关键词的相关度，由于它们过于包容潜在的（不正确的）兴趣词项，通常会造成一些错误匹配。通过与基于分类的过滤器（或提升）组合来解决这个问题，会将最佳匹配结果限制在已知良好的通用分类中，由于最佳结果匹配了文档的具体特征，因而相关度得到了提升。这也有助于克服分类推荐方法的局限，单独使用分类时只能匹配较宽泛的类目。

协同过滤擅长寻找基于用户行为的文档关系，但有时仅推荐其他用户感兴趣内容的会忽略当前用户的个性化特征。在查询的协同过滤子句添加基于属性的过滤器，可以更有针对性地向当前用户推荐（与其他用户达成一致的）相关文档。如果所有用户都对位置敏感，可以通过添加地理位置子句来改进这些方法。在一个查询中组合多种方法会影响搜索性能（速度），这一点要谨记。即便如此，多数行业研究发现，与单一模型相比，多种方法组合使用会得到更好的搜索结果。如果你正在寻找一个良好的开端，并且也拥有数据，那么在多数现实应用中协同过滤要优于基于内容的推荐方法，因此可优先考虑。从协同过滤入手后，你还可以不断添加其他方法，为用户提供不断改进的推荐体验。



## 16.6 塑造个性化搜索体验

上一小节详细介绍了如何使用 Solr 构建一个推荐引擎，实现自动搜索出相似内容的功能。这包括利用用户行为信息来提升相似文档的相关度权重，以及根据用户和他们可能会搜索的内容之间的已知相似度来匹配文档的内容。

虽然搜索引擎和推荐引擎通常会被认为是两码事，许多公司将两者作为独立的技术体系来运行，但如果将搜索与推荐作为匹配的互补方式，将会带来相关度改进的多种可能。

本书介绍了用户搜索体验质量的多种改进方法。搜集用户与文档的有关信息，包括用户之前搜索中表达的兴趣、关键词和搜索位置，通过注册信息和个人资料来搜集用户的兴趣——我们已经看到了如何通过充分了解用户来实现自动化推荐。

读到这里，不妨考虑整合这些方法，通过行为数据来影响传统的搜索结果。那么，何不考虑让用户自己选择使用过滤器，进一步对他们的推荐查询进行实时优化呢？如果用户在搜索应用中执行关键词搜索，但没有指定地点，通过添加“距离提升因子”（参见 16.5.6 节）可以轻松地对距离用户较近的文档进行提升。同样，如果用户搜索特定位置附近的文档（例如，餐馆搜索、求职搜索或演出搜索），则提升那些与用户资料中表达的兴趣相匹配的文档。此外，如果能获取用户的 IP 地址，则可以根据 IP 地址所在位置自动提升其附近的文档。值得一提的是，这种做法取决于数据情况以及呈现方式。如果没有得到用户默许而擅自使用他们的个人信息，个性化搜索体验可能会让用户感到后背发凉。隐私是用户关心的一个切实问题，因此对提升搜索用户体验的个性化方式要谨慎考虑。

根据搜集到的或学习到的用户信息来提升个性化相关度，将这种方法与传统搜索体验相结合，用户可以更快地找到符合其特定兴趣的相关信息。这种方法并不适用于所有的搜索应用类型，但许多大型网络搜索引擎都在向个性化搜索方向发展，开展了大量此项业务。如果搜索应用的一个重要目标是为用户提供相关的内容；通过加入用户信息来增强搜索体验是可行的，而且借助本章介绍的 Solr 内置功能与技术实现起来也并不那么困难。

相关度改进的每个想法不一定都能奏效。可能需要尝试许多技术及各种配置，才能达到所需的相关度要求。事实上，许多机构会把相关度的调整作为一项长期事业，不断尝试改进。下一节介绍如何自行开展这类实验。

## 16.7 开展相关度实验

本章主要介绍了 Solr 搜索应用的相关度扩展方法。本章提出的这些技术理念实现起来需要很长时间，我们期望它们能够改进用户的搜索体验质量，那么如何对改

进效果进行实际评测呢？

如果搜索应用有特定的业务指标，无疑从这些业务指标着手会是不错的选择。如果是电子商务应用，商品销售总额就是一个业务指标；如果是网络搜索引擎，平均倒数排名（Mean Reciprocal Rank, MRR）可能是评价指标（[http://en.wikipedia.org/wiki/Mean\\_reciprocal\\_rank](http://en.wikipedia.org/wiki/Mean_reciprocal_rank)），它评价了搜索结果中理想结果的靠前程度，或用户在找到搜索结果之前所点击的链接平均数；如果是求职搜索引擎，评价指标可能是求职者如何快速找到一份工作，或根据算法产生的额外职位数。当使用业务指标进行评价时，即使最终用户可能不赞同该指标，还是要根据搜索引擎的特定目标来定义相关度。从商业角度来看，假如你选择了正确的指标，这就已经达到了预期结果。

在这种情况下最佳方式是尝试不同的相关参数，进行 A/B 测试，在同一时间段随机将用户分组，对每一组被试实验一种算法。假设要评测两种新算法，每种算法测试 10% 的用户。将 10% 的用户放在一个控制组，用于当前算法；10% 的用户放在一个测试组，用于算法 1；10% 的用户放在另一个测试组，用于算法 2。这样可以分别对独立的三组进行指标评测。除非使用很好的开源或商业框架进行实验，否则我们肯定希望掌握基本的统计信息，以确保评价结果统计是有效的。

为了搜集结果，A/B 测试需要向一部分最终用户说明变化。除此之外，算法相对性能评估的另一种常见方法是使用已有的日志数据：生成一个查准率与查全率图表。在这种方法中，我们可以在日志文件中保存的用户行为数据上测试每一个候选算法，可以在对用户实际在用之前预测这些搜索结果对用户的影响。在电子商务平台中，为用户执行每个查询或推荐，得到搜索结果（假设每个搜索请求得到 5 个结果），将它们绘制在查准率与查全率图上，根据用户的历史行为判断算法是否做出了正确的预测。例如，在电子商务平台中，一个正确的预测可能是找到用户购买过的商品。因此，任何能够在搜索这些商品时得到高查准率与查全率的查询模型，就可以被视为一个不错的算法。表 16.2 展给出了一些查准率与查全率数据。

表 16.2 根据多个搜索的基准与历史用户数据进行比照的查准率与查全率表

结果 #	当前算法		算法 1		算法 2	
	查准率	查全率	查准率	查全率	查准率	查全率
1	0.8	0.3	0.65	0.25	0.84	0.31
2	0.56	0.41	0.51	0.35	0.65	0.38
3	0.45	0.5	0.43	0.43	0.51	0.49
4	0.35	0.63	0.33	0.55	0.41	0.62
5	0.28	0.8	0.26	0.76	0.3	0.79

表 16.2 的数据是基于历史用户信息的查准率与查全率计算值。这里的假设是，



匹配到用户之前采取行动的文档就是正确的匹配。第3章的3.3节提到过，查准率 = 正确的匹配结果数 / 所有返回的结果数，查全率 = 正确的匹配结果数 / (正确的匹配结果数 + 未匹配到的结果数)。查准率与查全率并不是完全负相关的，虽然两者之间存在此消彼长的互逆关系。表16.2的数据很容易转换成图表，如图16.8所示。

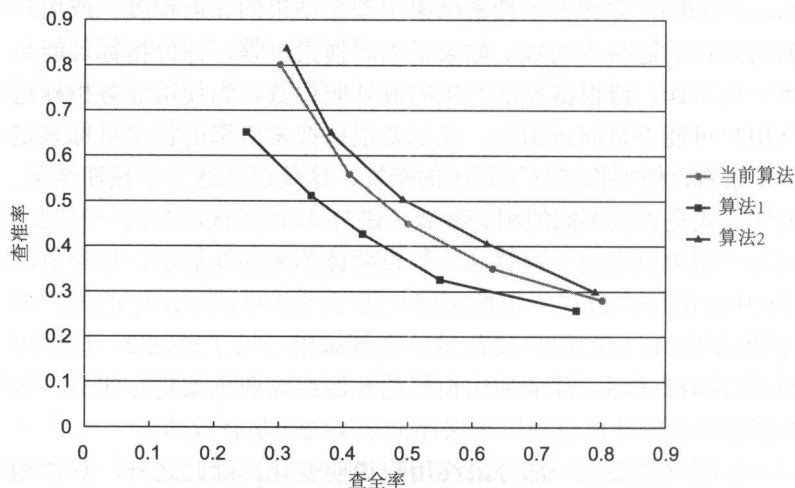


图 16.8 查准率与查全率图显示了不同相关度算法的相对性能，历史数据反映了用户实际偏好的内容，以此作为评测标准

从图16.8可以看出，算法2拥有更高的查准率与查全率，表示其性能优于当前算法与算法1。进行这类分析的一个好处在于，分析结果完全基于历史基准数据生成，这意味着，测试可以在离线环境下运行，而不会对生产系统造成影响。另外，还有其他一些指标，如F值(F-measure, [http://en.wikipedia.org/wiki/Precision\\_and\\_recall#F-measure](http://en.wikipedia.org/wiki/Precision_and_recall#F-measure))，它会计算出一个绝对值，综合了查询率和查准率两个指标。

对图16.8的曲线进行分析，不管是从单一值(如F值)角度，还是从查准率与查全率两个角度来看，离线相关度测试都能够对算法进行合理检测，降低对生产系统的A/B测试耗时，最大限度地减少软件缺陷导致的意外问题，而不是等到测试算法时才发现与真实用户期望相悖。经由A/B产品测试和离线查准率/查全率测试，可以确保一定精确度下遵守恰当的统计准则。在调整算法时，算法测试应该只在一部分历史数据上进行，例如，对汇总的历史数据不要使用参数。如果用户对算法的反应存在较大差异，那么即使一个算法看起来优于另一个，结果也可能是站不住脚的。所以在这种情况下应该进行统计学意义上的显著性检验。读到这里，你应该已经了解如何通过启用多种功能、使用数据以及调整变量来改进搜索应用的相关度。随着对Solr的深入理解和相关度检验的实践，你应该有能力创建高度相关的搜索体验了。

## 16.8 本章小结

恭喜你！你已经读完了《Solr 实战》的最后一章！你不仅学会了如何像专家一样配置与运行 Solr，还见识了许多应用实例。这些实例代表了 Solr 大部分核心功能的最佳实践，从对内容进行文本分析配置到分组、分面以及复杂的数据操作，Solr 让你可以在横向可扩展和高吞吐量的搜索服务器上向客户提供实时且相关的结果。无论你是否需要处理复杂的多语言文档，相信你已经学会了如何实现自动建议或拼写纠错系统，或简单地在搜索中仔细“打磨”相关度，这会让你更有信心去解决一些具有挑战性的搜索难题。

希望你能够从本书的概念探讨和应用实例中学到很多东西。如果打算使用 Solr 构建一些奇妙的应用，希望你可以做到最好，让梦想照进现实，助力你的职业生涯发展，未来影响到整个世界，加油！

# 与Solr代码库打交道

Solr 与 Lucene 官方每年会发布若干个新版本。得益于完全开源，我们随时可以获取与编译 Solr 的新近主干版本。Lucene 与 Solr 的代码库存在关联，也就是说，即使没有 Solr，Lucene 也可以单独使用；另外，Solr 的开发进度与 Lucene 同步，两者的代码库同步进行编译与正式对外发布。Lucene/Solr 的主干版本包含所有已提交的更改，不过，在主干版本没有成为官方版本之前，它的任何功能仍有可能发生变化。

除了新提交的更改，Solr 的 JIRA 页面 (<https://issues.apache.org/jira/browse/SOLR>) 罗列了世界各地的开发人员贡献的大量补丁。一旦 Solr 提交者同意并赞成补丁里贡献的更改，许多补丁最终就会被提交到 Lucene/Solr 的主干版本。所有代码都是公开的，可供自由使用。因此，如果官方版本里没有你需要的功能或某些补丁还未被修复，你就可以定制自己的 Solr 分发版本。本节介绍 Solr 定制分发版本的开发与部署过程。

## A.1 获取Solr的正确版本

开发 Solr 定制分发版本的第一步是，获取 Lucene/Solr 的相应版本作为基础。你可以从 Solr 主页 (<http://lucene.apache.org/Solr/>) 下载完整的源代码和二进制文件 (参见第 2 章)，或者使用 SVN 或 Git 获取，具体操作命令如下：

```
svn checkout http://svn.apache.org/repos/asf/lucene/dev/trunk  
➡ lucene-solr
```

或者

```
git clone https://github.com/apache/lucene-solr.git lucene-solr
```

SVN 仓库是 Lucene/Solr 的官方代码库。喜欢使用 Git 版本控制系统的读者可以使用 Github 仓库镜像（存在轻微延迟）。对于 SVN 版本，你必须在取出代码时指明想要获取的 Solr 的具体版本。主干分支是活跃的开发试验田，可视为仍处于试验中的最新版本。若想获取非当前主干版本，则需要在 SVN 的 URL 中用特定版本的标签或分支替代“trunk”。官方版本表示为 *tags/lucene\_solr\_x\_y\_z*（其中 *x\_y\_z* 表示发布的版本号），因此，获取 Solr 4.7.0 官方版本的操作命令如下：

```
svn checkout  
➡ http://svn.apache.org/repos/asf/lucene/dev/tags/lucene_solr_4_7_0  
➡ lucene-solr
```

同样地，若想获取 Solr 的当前开发分支（不一定被标记了版本号），则需要在 SVN 的 URL 里用“branches”替换“tags”，指定想要获取的分支名：

```
svn checkout  
➡ http://svn.apache.org/repos/asf/lucene/dev/branches/lucene_solr_4_7  
➡ lucene-solr
```

若使用 Git，可以通过之前的 `Git clone` 命令获取整个 Solr 仓库（其中包含了所有版本），然后使用以下命令切换到特定版本（例如，Solr 4.7.0）：

```
cd lucene-solr  
git checkout tags/lucene_solr_4_7_0
```

若要查看当前镜像版本（本书付印时是 Solr 4.7.x）的活跃开发分支，可以使用以下命令获取相应分支：

```
git checkout lucene_solr_4_7
```

在之前的两个 Git 示例中，第一个示例会切换到 Lucene/Solr 4.7.0 官方标记版本，第二个示例会获取 4.7.x 版本的开发分支。若想浏览可用的分支与标记的发行版本，可以访问 Github（<https://github.com/apache/lucene-solr>）页面的版本列表，或使用从 Solr 官方网站的版本控制页面（<http://lucene.apache.org/solr/versioncontrol.html>）链接到 Apache 软件基金会的官方 SVN 网络界面。

## A.2 在IDE中构建Solr

Solr 的专业开发需要在集成化开发环境 (Integrated Development Environment, IDE) 中安装 Lucene/Solr。由于获取的 Solr 的具体版本不同, 对 Java 最低版本的要求可能存在略微差异, 这里假定你已经安装 Java 1.7 JDK。如果有需要, 可以通过 Google 查找你的操作系统对应的 Java 安装指南。本节介绍如何在 Eclipse 与 IntelliJ IDEA 中安装 Solr, 这是 Solr 开发最常见的两种免费 IDE。

当获取 (下载与解压缩) 代码到 lucene-solr/ 目录之后, 只需几个步骤便可以在你喜欢的 IDE 里安装 Solr。在创建的 lucene-solr/ 目录中有很多内容, 其中包含三个重要文件: lucene/ 目录 (包含 Lucene 的所有源代码)、solr/ 目录 (包含 Solr 的所有源代码) 与 build.xml 文件 (包含 Ant 编译系统所有任务)。

### Ant、Ivy 与外部依赖

Lucene 与 Solr 组合代码库使用 Apache 的 Ant 作为编译系统。Ant 是一个基于 Java 的命令行工具。全局 build.xml 文件作为整体位于代码库的主目录。除了主目录之外, Lucene 与 Solr 分别在 lucene/ 与 solr/ 目录下各自包含单独的 build.xml 文件。

由于 Lucene 与 Solr 包含许多外部库依赖, 而且这些库可能也被其他项目开发正在使用, 所以开发者在一开始下载 Solr 时, 需要把所有相关的依赖库一并下载下来, 这很不方便。为了解决这个问题, Lucene 与 Solr 代码库使用 Apache 的 Ivy 进行依赖关系管理。当编译 Lucene 与 Solr 时, Ivy 会检查库的依赖关系, 若之前没有获取过的库, 则 Ivy 会从公开的可信来源处下载该库。

如果使用 Ant, 但未曾安装 Ivy, 编译 Solr 时可能会出现 Ivy 不可用的异常情况。若出现此类情况, 你应该在 Lucene/Solr 代码库的主目下执行以下命令:

```
ant ivy-bootstrap
```

该命令会自动安装 Ivy, 确保能够自动获取 Solr 成功编译所需的任意外部依赖库。若计划在 IDE 中安装 Solr, 则首先需要确认已经安装了 Ivy, 并且至少编译过一次 Lucene/Solr (编译时所有依赖库会自动获取), 以确保 IDE 中不会出现“缺少依赖关系”问题。在 lucene-solr/ 目录下执行以下命令来编译 Solr:

```
cd solr/  
ant dist
```

虽然 Lucene 和 Solr 的代码库与 IDE 没有直接关系, 但它为 Eclipse 与 IntelliJ IDEA 这两个最流行的 Lucene/Solr 开发 IDE 提供了方便的配置文件创建命令。

## 在 Eclipse 中导入 Lucene/Solr

为了确保 Eclipse 的无缝导入,请在 Lucene/Solr 代码库的主目录下执行以下命令:

```
ant eclipse
```

配置文件已准备就绪,接下来打开 Eclipse,导入 Lucene/Solr 的 project 目录(即 A.1 节中获取 Solr 时的 lucene-solr/),创建一个项目。这些操作在 Eclipse 的 Kepler 版本上创建,不做任何改动,它们应该也可以在新近的 Eclipse 版本上执行。打开 Eclipse 之后,选择 File 菜单,点击“导入”,选择“常规”>“现有项目”,进入到“工作区”>“下一步”,选择根目录选项,输入或浏览定位到 lucene-solr/ 目录所在的路径,最后点击完成按钮。至此, Lucene/Solr 项目已经成功导入,可以开始开发了!

## 在 IntelliJ IDEA 中导入 Lucene/Solr

使用 IntelliJ IDEA 开发或调试 Solr 代码,导入步骤与在 Eclipse 中的导入过程类似。此步骤在 IntelliJ IDEA 12 社区版上创建,在其他新近的版本上只需做细微改动。初始创建 IntelliJ IDEA 必要的配置文件,需要在 lucene-solr/ 目录下执行相同的命令:

```
ant idea
```

当 IntelliJ IDEA 配置文件准备就绪,打开 IDE,选择文件菜单,点击导入按钮,导航到你的 lucene-solr/ 目录,然后点击下一步按钮。从已知来源选项创建新项目,再次点击下一步按钮。接下来的几个窗口显示若干选项,比如,重新调整项目外观,选择 Java JDK,以及选择库与依赖关系。一般来说使用默认选项,点击下一步按钮,直到安装完成。

恭喜!你已经成功导入 Lucene/Solr 项目,现在可以在 IntelliJ IDEA 中进行开发了。进行开发时,重要的一点是能够调试任意更改,下一小节会详细介绍。

## A.3 Solr代码调试

调试 Solr 代码库(以及你对 Solr 做出的所有更改)的最简单方法是,在启动 Solr 的示例应用时,启用 JVM 远程调试功能。该方法适用于大多数 IDE,方便从远程服务器上调试 Solr。修改 Solr 代码(或添加你自己的插件)之后,要编译 Solr 示例应用的最新版本,在上一节创建的 lucene-solr/ 根目录下执行以下命令:

```
cd solr/  
ant example
```

该命令完成 Solr 与示例 Web 应用的重编译之后,在启动 Solr 时,可以通过参数传递启用远程调试:



```
cd example/  
java  
➤ -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=8984  
➤ -jar start.jar
```

agentlib:jdwp 参数在远程调试模式下启动 JVM。远程调试端口地址可以任意指定。我们选择一个与 Solr 默认端口接近的端口号。suspend=y 选项会告诉 JVM，在没有添加调试器之前，不要执行 start.jar 代码。若需要 Solr 以正常方式启动，以后再添加调试器的话，设置 suspend=n。若需要调试 Solr 的启动逻辑，则设置 suspend=y。有时向 Solr 发送一个请求之前想要进行调试，晚些时候再添加调试器可能更容易些。

### 在 Eclipse 里运行 Solr

如果你使用 Eclipse，并且不需要远程调试 Solr 的话，另一种方法是，通过 Solr 包含的 start.jar，在 Eclipse 里直接将 Solr 运行在 Jetty 上。Solr 的输出会被发送到 Eclipse 的控制台，这样 Solr 就可以完全运行在 Eclipse 里了。

创建一个调试配置，点击“运行”>“调试配置”>“Java 应用”，设置如下：

主类（主选项）：org.eclipse.jetty.start.Main

工具目录（参数选项）：\${workspace\_loc:lucene-solr/ solr/example/}

然后，将 Solr 的 start.jar 文件添加到 classpath，这样 Eclipse 就能够找到并执行 Jetty 的 Main 类。选择 Classpath 选项，点击如下选项。

Classpath > User Entries > Add JARs > lucene-solr/solr/example/start.jar

这样就设置好了。保存这个配置文件，以后在 Eclipse 中调试 Solr 时可以无限地重复使用。大多数其他 IDE 中的配置文件创建方法类似。若将 Solr 运行在 IDE 的外部，你也可以使用之前介绍的远程调试配置。下一节将介绍如何调试该配置。

### 将 IDE 连接到正在运行的 Solr 实例

Solr 以远程调试模式启动后，可以将 IDE 连接到正在运行的 Solr 实例。在 Eclipse 里，选择菜单栏，点击“运行”>“调试配置”>“远程 Java 应用”，将 lucene-solr 视为一个项目，localhost 作为主机，8984 作为端口，然后点击调试。

若之前设置了 suspend=y，就可以从启动调试模式的终端窗口看到 Solr 启动，接下来在 Eclipse 里设置断点，逐步调试 Solr 已经运行的代码。

若使用 IntelliJ IDEA，则操作步骤类似。在菜单栏中选择“运行”>“编辑配置”，



然后点击“+”按钮，选择远程选项作为想要添加的配置新类型。接下来，为该配置命名（如 lucene-solr），主机为 localhost，端口为 8984，点击“OK”按钮。调试配置定义好之后，返回菜单栏，选择“运行”>“调试选项”，然后选择 lucene-solr（之前定义的调试配置名）进行调试。

### 远程调试 Solr 代码

值得注意的是，虽然我们刚才使用这项技术对 solr 进行了本地调试，但它其实也具备在网络计算机中进行远程调试的能力。这意味着，如果手头有一台启用了远程调试的 Solr 生产服务器，你就可以把 IDE 与远程服务器连接起来，以调试遇到的任何代码问题。你需要做的是在 IDE 的调试配置中更改服务器名称，将 localhost 改为服务器所在网络的地址，确保远程调试端口没有被网络封锁。

至此，你已经了解如何在 IDE 中安装 Solr 从而为开发做好准备，以及如何调试 Solr 代码。接下来向 Solr 提供代码补丁，添加其他功能或修正已知错误，从而改进 Solr 的分发版。

## A.4 Solr补丁下载与应用

有时你可能会发现 Solr 的一个错误，或者某个你所需要的而 Solr 还未提供的功能。因为 Solr 是一个众多开发者共同参与的开源项目，所以其他开发者可能也有相同的需求，甚至已经开始编码来解决这个问题了。这是常有的事情。Solr 的公开问题跟踪单系统详见 <https://issues.apache.org/jira/browse/SOLR>。

这个问题跟踪单系统（以下称为 Solr JIRA）以补丁文件形式向 Solr 代码库提交功能请求与更改。在这里你可能会找到能解决你的问题的代码，尽管这些代码可能还未完成。

打补丁要找到最新可用的补丁文件。请记住一点，若补丁文件不符合你的 Solr 版本，打补丁时可能会由于代码不匹而配导致出错。

Solr 社区使用的补丁是由 `svn diff` 或 `git format-patch` 命令（或其他版本控制系统中的 `diff` 格式）生成的标准补丁文件。对 `svn diff` 命令生成的补丁进行测试时，在 lucene-solr 目录下执行以下命令：

```
wget <URL to the patch> -O - | patch -p0 --dry-run
```

类似地，如果补丁是由 `git format-path` 命令生成的，由于补丁格式不同，你需要传递 `-p1` 选项而不是 `-p0` 选项。

```
wget <URL to the patch> -O - | patch -p1 --dry-run
```

假定两个示例都想要通过 Solr JIRA 上的 URL 获取已公开发布的补丁，而不必将补丁手动下载到计算机上。不过，在本地补丁文件上测试补丁也是可以的。如果你已经准备好补丁文件的本地副本，则需要将它复制到 Lucene/Solr 项目的根目录(之前获取的 lucene-solr/ 目录)，并执行如下 SVN 样式的补丁命令。

```
patch -p0 -i <patchfile> --dry-run
```

如果打补丁遇到问题，可以尝试以下命令，补丁会被转换为默认的 Git 补丁样式：

```
patch -p1 -i <patchfile> --dry-run
```

这些补丁命令包括 --dry-run 标记，它能够测试补丁是否可以完全打上去，而不必真正应用补丁。在补丁应用之前测试一下它通常是最佳实践，可以避免因为补丁测试而导致代码库不完整的情况。补丁测试通过后，需要从命令里移除 --dry-run 参数。

如果打补丁过程中出错了，或想要恢复之前状态，可以使用 `svn revert -R ./` 或 `git reset --hard HEAD` 命令恢复仓库。代码控制系统的操作有很多技巧（这超出了本书的讨论范围），以实现在不损坏本地仓储的前提下，支持本地分支中前后切换，方便地进行更改测试。这些补丁选项提供了你所需要的、社区贡献的可供使用的补丁。

测试补丁之后，你可能想要贡献自己的改进。下一节介绍如何创建与提交一个 Solr 补丁。

## A.5 贡献补丁

如果你决定修正 Solr 的一个错误、添加一个功能，或改进已有的补丁，你可以将它上传到 Solr JIRA，供其他人使用。这时需要创建一个 Solr JIRA 账号，如果你要贡献的功能在 Solr 中尚不存在，则可以创建一份追踪单。

如果你以为需要提交一份新的 Solr JIRA 追踪单，最好先发邮件到 Solr 用户邮件列表，来确保你要贡献的功能确实是 Solr 中尚不存在的，或者发邮件到 Solr 开发者邮件列表，确保你的做法处在正确的解决方向上。邮件列表注册相关信息详见 <http://lucene.apache.org/solr/discussion.html>。

当你确定某个补丁是必要的，并且编写代码解决了该问题后，还需注意以下几点。首先，Solr 维护了一个编码指南 ([http://wiki.apache.org/solr/HowToContribute#Making\\_Changes](http://wiki.apache.org/solr/HowToContribute#Making_Changes))，所有提交的补丁都应遵守。对新补丁而言，最重要的一点是，不能破坏 Solr 或 Lucene 已有的单元测试。开始创建补丁之前，你应该在 lucene-

solr/ 目录下执行完整的单元测试。

```
ant clean test
```

如果测试运行的输出显示 BUILD SUCCESSFUL (编译成功), 就可以继续创建补丁, 否则就要检查测试失败的原因, 及时纠正错误。为避免其他基础错误, 在创建补丁文件之前, 请运行 precommit Ant 任务。

```
ant precommit
```

如果所有的 precommit 检查都通过了, 就可以开始创建补丁了。如果你正在使用 SVN, 则需要找到所有修改过的文件:

```
svn stat
```

接下来, 根据 svn stat 命令得到的清单, 添加新文件:

```
svn add path/to/newfile1
svn add path/to/newfile2
...
```

Solr 与 Lucene 在它们的项目根目录 (solr/ 与 lucene/) 下都维护了一个 CHANGES.txt 文件。在贡献补丁之前, 你应该向 Solr 的 CHANGES.txt 文件中添加一条更改信息, 包括 JIRA 跟踪单号与更改的简要说明。如果你更改了 Lucene 文件, 也应该在 Lucene 的 CHANGES.txt 里添加一条信息。一旦更改被记录, 最后一步是执行 svn diff 命令, 创建补丁文件:

```
svn diff > SOLR-XXXX.patch
```

这条命令根据补丁单号创建补丁, 然后上传到 Solr JIRA。补丁文件名应遵守特定的格式, 未来的所有更新补丁都应以相同的文件名上传, 告知 Solr JIRA, 这些补丁文件只是同一个补丁的更新版本。

如果你使用的是 Git, 使用以下 Git 命令创建补丁:

```
git status
git add path/to/file1
git add path/to/file2
...
git commit -m "Description of your commit"
git format-patch --stdout -p --no-prefix -1 > SOLR-XXXX.patch
```

关于 git format-patch 命令需要注意两点。首先, -1 参数表示补丁应该根据与之前最后一次提交的差异进行创建。如果尝试创建多个提交的补丁, 最好使用你更改之前的最后一个提交的 ID 替换 -1。可以使用 git log 命令可以查看最近的提交记录, 以查找适合的提交 ID。

其次, `git format-patch` 命令是 `--no-prefix` 参数, 这会导致补丁文件以 `patch -p0 format` (SVN 样式) 生成, 而不是以 `patch -p1` 格式 (默认 Git 样式) 生成, 参见上一小节讨论的打补丁。

由于 Solr 社区同时使用 SVN 与 GIT, 因此前一种补丁格式 (`patch -p0`) 是更有用的标准格式, 更容易在两种系统之间协同。

当创建好补丁之后, 你可能会考虑恢复本地仓库, 并按照 A.4 节的步骤打补丁, 以确保补丁没有遗漏任何更改。如果所有步骤都通过了, 最后一步是根据你创建的或与他人合作创建的补丁单号, 将补丁上传到 Solr JIRA。如果一切顺利, 而且 Solr 贡献者同意了你的更改, 很快就可以在 Solr 的官方版本里看到你的贡献了! 所有补丁完成并在 Lucene/Solr 代码库更新之后, 就可以编译与部署 Solr 了。请参见第 12 章来了解如何在生产环节中编译与部署 Solr。

# B

## 语种字段类型配置

由于语种之间的巨大差异，在 Solr 中为不同语种定义字段时没有统一的模型可供参考。一些语种需要特定的词干提取过滤器，另一些语种则需要多个过滤器来处理不同的语言特征（如规范字符、去除口音及自定义小写功能等）。由于语种解析的复杂性，一些语种甚至需要自己的分词器。下表为 Solr 支持的多种语言的开箱即用配置，它们虽然不是语种配置的唯一方法，但可以视为一个好的切入点。

从表中可以看出，许多分析组件在多个语种之间共享，特定语种的组件与配置粗体标记以示区别。

语 种	分析器链条举例
阿拉伯语	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/ <b>stopwords_ar.txt</b> "]
	<b>ArabicNormalizationFilterFactory</b>
	<b>ArabicStemFilterFactory</b>
亚美尼亚语	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/ <b>stopwords_hy.txt</b> "]
	SnowballPorterFilterFactory [language=" <b>Armenian</b> "]

续表

语 种	分析器链条举例
巴斯克语	StandardTokenizerFactory LowerCaseFilterFactory StopFilterFactory [words="lang/ <b>stopwords_eu.txt</b> "] <b>SnowballPorterFilterFactory</b> [language="Basque"]
保加利亚语	StandardTokenizerFactory LowerCaseFilterFactory StopFilterFactory [words="lang/ <b>stopwords_bg.txt</b> "] BulgarianStemFilterFactory
加泰罗尼亚语	StandardTokenizerFactory ElisionFilterFactory [articles="lang/ <b>contractions_ca.txt</b> "] LowerCaseFilterFactory StopFilterFactory [words="lang/ <b>stopwords_ca.txt</b> "] SnowballPorterFilterFactory [language="Catalan"]
中文	<b>SmartChineseSentenceTokenizerFactory*</b> <b>SmartChineseWordTokenFilterFactory*</b> LowerCaseFilterFactory PositionFilterFactory * 注意：此处需要额外的依赖包。详细说明参见 <i>solr/contrib/analysis-extras/README.txt</i>
CJK (中文、日文、韩文二元文法)	StandardTokenizerFactory <b>CJKWidthFilterFactory</b> LowerCaseFilterFactory <b>CJKBigramFilterFactory</b>
捷克语	StandardTokenizerFactory LowerCaseFilterFactory StopFilterFactory [words="lang/ <b>stopwords_cz.txt</b> "] <b>CzechStemFilterFactory</b>
丹麦语	StandardTokenizerFactory LowerCaseFilterFactory StopFilterFactory [words="lang/ <b>stopwords_da.txt</b> "] SnowballPorterFilterFactory [language="Danish"]
荷兰语	StandardTokenizerFactory LowerCaseFilterFactory StopFilterFactory [words="lang/ <b>stopwords_nl.txt</b> "] StemmerOverrideFilterFactory [dictionary="lang/ <b>stemdict_nl.txt</b> "] SnowballPorterFilterFactory [language="Dutch"]

续表

语 种	分析器链条举例
英语	StandardTokenizerFactory
	StopFilterFactory [words="lang/stopwords_en.txt"]
	LowerCaseFilterFactory
	<b>EnglishPossessiveFilterFactory</b>
	( <b>EnglishMinimalStemFilterFactory</b>
	或
芬兰语	<b>KStemFilterFactory</b>
	或
	<b>PorterStemFilterFactory</b>
	或
	SnowballPorterFilterFactory) [language="English"]
	StandardTokenizerFactory
法语	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_fr.txt"]
	( <b>FinnishLightStemFilterFactory</b>
	或
	SnowballPorterFilterFactory) [language="Finnish"]
	StandardTokenizerFactory
加利西亚语	ElisionFilterFactory [articles="lang/contractions_fr.txt"]
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_fr.txt"]
	( <b>FrenchLightStemFilterFactory</b>
	或
	<b>FrenchMinimalStemFilterFactory</b>
加利西亚语	或
	SnowballPorterFilterFactory) [language="French"]
	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_gl.txt"]
	( <b>GalicianStemFilterFactory</b>
加利西亚语	或
	<b>GalicianMinimalStemFilterFactory</b> )



续表

语 种	分析器链条举例
德语	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_de.txt"]
	<b>GermanNormalizationFilterFactory</b>
	( <b>GermanLightStemFilterFactory</b>
	或
希腊语	<b>GermanMinimalStemFilterFactory</b>
	或
	SnowballPorterFilterFactory) [language="German"]
	StandardTokenizerFactory
	GreekLowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_el.txt"]
北印度语	<b>GreekStemFilterFactory</b>
	StandardTokenizerFactory
	LowerCaseFilterFactory
	<b>IndicNormalizationFilterFactory</b>
	<b>HindiNormalizationFilterFactory</b>
	StopFilterFactory [words="lang/stopwords_hi.txt"]
匈牙利语	<b>HindiStemFilterFactory</b>
	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_hu.txt"]
	( <b>HungarianLightStemFilterFactory</b>
	或
印度尼西亚语	SnowballPorterFilterFactory) [language="Hungarian"]
	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_id.txt"]
	<b>IndonesianStemFilterFactory</b>
	StandardTokenizerFactory
爱尔兰语	ElisionFilterFactory [articles="lang/contractions_ga.txt"]
	StopFilterFactory [words="lang/hyphenations_ga.txt"]
	<b>IrishLowerCaseFilterFactory</b>
	StopFilterFactory [words="lang/stopwords_ga.txt"]
	SnowballPorterFilterFactory [language="Irish"]

续表

语 种	分析器链条举例
意大利语	StandardTokenizerFactory
	ElisionFilterFactory [articles="lang/contractions_it.txt"]
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_it.txt"]
	(ItalianLightStemFilterFactory 或 SnowballPorterFilterFactory) [language="Italian"]
日语	JapaneseTokenizerFactory [userDictionary="lang/userdict_ja.txt"]
	JapaneseBaseFormFilterFactory
	JapanesePartOfSpeechStopFilterFactory [tags="lang/stoptags_ja.txt"]
	CJKWidthFilterFactory
	StopFilterFactory [words="lang/stopwords_ja.txt"]
拉脱维亚语	JapaneseKatakanaStemFilterFactory
	LowerCaseFilterFactory
	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_lv.txt"]
挪威语	LatvianStemFilterFactory
	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_no.txt"]
	(NorwegianLightStemFilterFactory 或 NorwegianMinimalStemFilterFactory 或 SnowballPorterFilterFactory) [language="Norwegian"]
波斯语	PersianCharFilterFactory
	StandardTokenizerFactory
	LowerCaseFilterFactory
	ArabicNormalizationFilterFactory
	PersianNormalizationFilterFactory
	StopFilterFactory [words="lang/stopwords_fa.txt"]

续表

语 种	分析器链条举例
葡萄牙语	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_pt.txt"]
	(PortugueseLightStemFilterFactory
	或
	PortugueseMinimalStemFilterFactory
罗马尼亚语	或
	PortugueseStemFilterFactory
	或
	SnowballPorterFilterFactory) [language="Portuguese"]
	StandardTokenizerFactory
	LowerCaseFilterFactory
俄语	StopFilterFactory [words="lang/stopwords_ro.txt"]
	SnowballPorterFilterFactory [language="Romanian"]
	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_ru.txt"]
	(RussianLightStemFilterFactory
西班牙语	或
	SnowballPorterFilterFactory) [language="Russian"]
	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_es.txt"]
	(SpanishLightStemFilterFactory
瑞典语	或
	SnowballPorterFilterFactory) [language="Spanish"]
	StandardTokenizerFactory
	LowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_sv.txt"]
	(SwedishLightStemFilterFactory
泰国语	或
	SnowballPorterFilterFactory) [language="Swedish"]
	StandardTokenizerFactory
	LowerCaseFilterFactory
	ThaiWordFilterFactory
	StopFilterFactory [words="lang/stopwords_th.txt"]
土耳其语	StandardTokenizerFactory
	TurkishLowerCaseFilterFactory
	StopFilterFactory [words="lang/stopwords_tr.txt"]
	SnowballPorterFilterFactory [language="Turkish"]

# 有用的数据导入配置

第 12 章介绍过，Solr 的数据导入处理器 DIH 提供了从多种外部数据源获取数据的能力。在第 10 章中，我们使用 DIH 将一部分维基百科数据转储文件中的页面转换成了 Solr 的文档，并且对它们进行了索引。本附录详细介绍如何配置 DIH 来启用导入功能，并演示如何导入完整的维基百科数据集与另一个适合试验的大型数据集：Stack Exchange 问答数据集。

## C.1 索引维基百科

在第 12 章中，我们将维基百科的一部分文档导入到预先配置好的 Solr 内核 solrpedia 中。为了启用 DIH，需完成以下几个步骤。

首先，我们需要获得维基百科数据集的一份副本。在第 10 章中，我们使用了维基百科的英文数据集中的一小部分文章（13 000 篇），包含在随书代码中。如果想尝试完整的数据集，请从 <http://dumps.wikimedia.org> 获得最新的维基百科数据集。请注意，下载的数据文件只包含每一篇文章的最新版本，解压后会占用 45 GB 的磁盘空间，更不要说在下载原始压缩文件（超过 10 GB）和对基构建的 Solr 索引了，可见数据量之庞大。附录提到的示例数据集可以在网站 `$SOLR_IN_ACTION/example-docs/ch10/documents/solrpedia.xml` 下载。

获取到数据集之后，要使用 DIH 功能还需修改三个重要配置。首先，在 `solrconfig.xml` 中添加 `<lib>`，为 DIH 及它的依赖项加载 JAR 文件。

```
<lib dir="../../contrib/dataimporthandler/lib" regex=".*\.jar" />
<lib dir="../../dist/" regex="solr-dataimporthandler-.*\.jar" />
```

其次，还需要在 `solrconfig.xml` 定义 `/dataimport` 请求处理器：

```
<requestHandler name="/dataimport"
  class="org.apache.solr.handler.dataimport.DataImportHandler">
  <lst name="defaults">
    <str name="config">data-config.xml</str>
  </lst>
</requestHandler>
```

注意，`DataImportHandler` 类从 `data-config.xml` 文件提取它的配置。设置 `data-config.xml` 是使 DIH 生效的第三步，也是配置的最后一步。代码清单 C.1 显示了导入维基百科的 XML 文章所需的 `data-config.xml` 文件配置元素。

### 代码清单 C.1 导入维基百科文章的数据导入配置

```
<dataConfig>
  <dataSource type="FileDataSource" encoding="UTF-8" />
  <document>
    <entity name="page"
      processor="XPathEntityProcessor"
      stream="true"
      forEach="/mediawiki/page/"
      url="solrpedia.xml"
      transformer="RegexTransformer,DateFormatTransformer">
      <field column="id"          xpath="/mediawiki/page/id"/>
      <field column="title"       xpath="/mediawiki/page/title"/>
      <field column="revision"    xpath="/mediawiki/page/revision/id"/>
      <field column="user"        xpath="/mediawiki/page/revision/contributor/username"/>
      <field column="userId"      xpath="/mediawiki/page/revision/contributor/id"/>
      <field column="text"        xpath="/mediawiki/page/revision/text"/>
      <field column="timestamp"   xpath="/mediawiki/page/revision/timestamp"
        dateTimeFormat="yyyy-MM-dd'T'hh:mm:ss'Z'"/>
      <field column="$skipDoc"    regex="^#REDIRECT .*"
        replaceWith="true" sourceColName="text"/>
    </entity>
  </document>
</dataConfig>
```

从本地文件中获取数据。

索引中的每一个页元素是一个文档。

从本地文件中加载 XML 数据。

索引文档的字段映射。

与指定的正则表达式相匹配的文档会被略过。

简言之，代码清单 C.1 上列出的配置会启用 DIH，解析 `solrpedia.xml` 文件中的 `<page>` 元素，创建索引文档。因此，需要将 `$SOLR_IN_ACTION/example-docs/ch10/documents/solrpedia.xml` 文件复制一份到 `$SOLR_INSTALL/example/` 下，这样 DIH 就能找到从哪里导入 XML 文件。如果你想知道数据在被导入之前是什么样子，请

在任意文本编辑器中查看 *solrpedia.xml* 文件。在第 10 章中，我们在运行过程中将 *solrpedia* 添加到 *solr* 实例中，这里使用内核管理页面能够在运行时从预先配置好的磁盘配置文件中创建 *Solr* 内核。如果 *Solr* 实例已经在运行，可以在配置文件与 *DIH* 设置好之后重启 *Solr*。

运行并且配置 *Solr* 内核以适应 *DIH* 之后，执行数据导入过程来创建索引。重新载入管理控制台，在 *solrpedia* 内核下选择 *Dataimport* 页面。填写表单，点击 *Execute* 按钮，如图 C.1 所示。

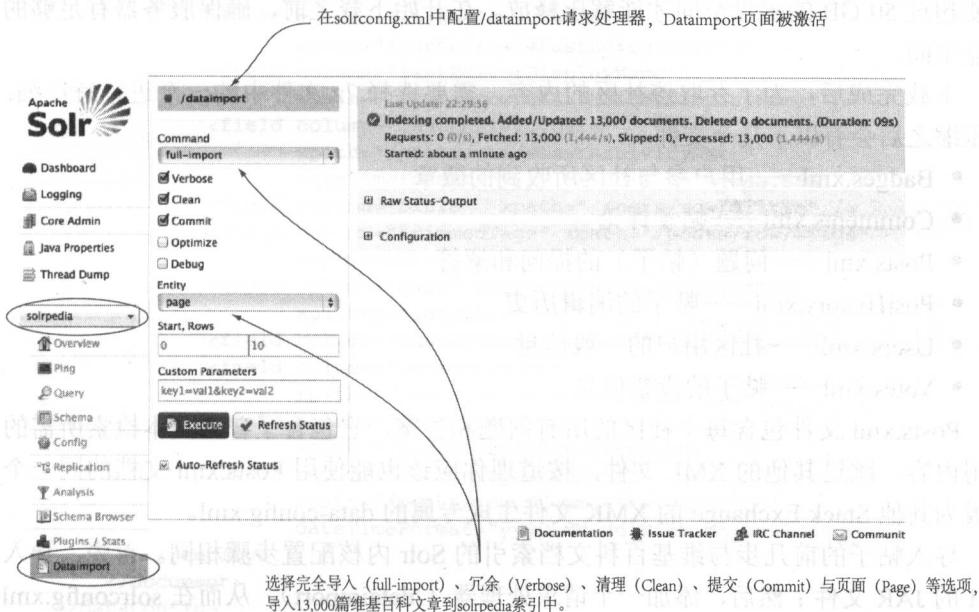


图 C.1 利用 *Dataimport* 页面从 *solrpedia.xml* 向 *solrpedia* 内核导入文章

有几个选项是可用的，包括是否完全导入、增量导入，在索引前是否清理之前的文档，在完成时是否提交或优化。另外，在导入过程中自动刷新状态选项会持续更新页面，以让你知道当前状况。如果一切顺利，你应该能看到文档的数量不断在攀升，直到维基百科数据文件的全部内容都索引到 *Solr* 内核中。

## C.2 索引Stack Exchange

除了维基百科文章，另一份公开可用的内容来自 *Stack Exchange* 问答数据文件的示例数据集。维基百科提供了关于重要实体的丰富的、描述性的（大部分）文章，而 *Stack Exchange* 数据是由许多不同类型话题的问答组成的。要创建 *Stack*

Exchange 的 Solr 索引，首先使用 BT 客户端从 <http://www.clearbits.net/creators/146-stack-exchange-data-dump> 下载一份月度数据文件。

数据文件下载完成后，你会看到许多压缩文件，每个文件对应 Stack Exchange 生态系统中不同子社区。例如，有些问答社区专注于科幻小说、游戏、烹饪，甚至是 SharePoint 和 WordPress，还有许多其他类型的主题社区。与维基百科数据文件类似，Stack Exchange 数据文件需要大量的磁盘空间。解压缩之前，下载的原始压缩包需要近 20 GB 的磁盘空间，解压之后最大的社区归档（Stack Exchange 的主社区）需要超过 50 GB 的磁盘空间才能解压释放。在开始下载之前，确保服务器有足够的磁盘空间。

下载完成后，为了存取该社区的内容，需要选择 7z 文件中的一个进行解压缩。解压缩之后会有以下几个文件：

- Badges.xml——用户参与社区所收到的徽章
- Comments.xml——帖子评论
- Posts.xml——问题（帖子）的提问和解答
- PostHistory.xml——帖子的编辑历史
- Users.xml——社区用户的一般信息
- Votes.xml——帖子的投票信息

Posts.xml 文件包含每个社区的所有问题和答案，它包含了普通文本检索所需的有用内容。跳过其他的 XML 文件，按道理你应该也能使用 Posts.xml 文档的同一个模板为其他 Stack Exchange 的 XMK 文件生成专属的 data-config.xml。

导入帖子的前几步与维基百科文档索引的 Solr 内核配置步骤相同。首先，导入 DIH 的 JAR 文件；然后，添加一个请求处理器（/dataimport），从而在 solrconfig.xml 启用 DIH。参见 C.1 小节的操作指南。

/dataimport 请求处理器配置好之后，创建 data-config.xml 文件，用于解析 Stack Exchange 帖子格式。代码清单 C.2 是导入一个 Posts.xml 文件的 data-config.xml 文件示例。

#### 代码清单 C.2 导入 Stack Exchange 帖子的数据导入配置

```
<dataConfig>
  <dataSource type="FileDataSource" encoding="UTF-8" />
  <document>
    <entity name="post"
      processor="XPathEntityProcessor"
      stream="true"
      forEach="/posts/row/"
      url="solrexchange.xml"
      transformer="RegexTransformer,
        DateFormatTransformer,HTMLStripTransformer">
```



```

<field column="id" xpath="/posts/row/@Id" />
<field column="postTypeId" xpath="/posts/row/@PostTypeId" />
<field column="acceptedAnswerId"
  xpath="/posts/row/@AcceptedAnswerId" />
<field column="creationDate" xpath="/posts/row/@CreationDate"
  dateTimeFormat="yyyy-MM-dd'T'hh:mm:ss.SSS" />
<field column="postScore" xpath="/posts/row/@Score" />
<field column="viewCount" xpath="/posts/row/@ViewCount" />
<field column="body" xpath="/posts/row/@Body"
  stripHTML="true" />
<field column="ownerUserId" xpath="/posts/row/@OwnerUserId" />
<field column="lastEditorUserId"
  xpath="/posts/row/@LastEditorUserId" />
<field column="lastEditorDisplayName"
  xpath="/posts/row/@LastEditorDisplayName" />
<field column="lastActivityDate"
  xpath="/posts/row/@LastActivityDate"
  dateTimeFormat="yyyy-MM-dd'T'hh:mm:ss.SSS" />
<field column="title" xpath="/posts/row/@Title" />
<field column="trimmedTags" xpath="/posts/row/@Tags"
  regex="&lt;(.*)&gt;" />
<field column="tags" sourceColName="trimmedTags"
  splitBy="&gt;&lt;" />
<field column="answerCount" xpath="/posts/row/@AnswerCount" />
<field column="commentCount"
  xpath="/posts/row/@CommentCount" />
<field column="favoriteCount"
  xpath="/posts/row/@FavoriteCount" />
<field column="communityOwnedDate"
  xpath="/posts/row/@CommunityOwnedDate"
  dateTimeFormat="yyyy-MM-dd'T'hh:mm:ss.SSS" />
</entity>
</document>
</dataConfig>

```

还应注意，data-config.xml 每个字段的列属性都要在 schema.xml 里进行定义。与第 10 章的 solrpedia 内核一样，我们创建了一个预先配置的 Solr 内核，称为 solrexchange，用来测试 Stack Exchange 帖子的导入。必要的字段配置文件 schema.xml 位于该内核的 conf/ 文件夹下。执行以下命令，创建 solrexchange 内核：

```

cd $SOLR_IN_ACTION/example-docs/appendixC
cp -r cores/ $SOLR_INSTALL/example/solr/
cp documents/solrexchange.xml $SOLR_INSTALL/example/
cd $SOLR_INSTALL/example/
java -jar start.jar

```

\$SOLR\_IN\_ACTION/example-docs/appendixC/documents/solrexchange.xml 文件包含 Posts.xml 文件的一个子集，Posts.xml 文件主要来自于 Stack Exchange 网站。具体来说，它只包含关于 Solr 或相关技术的帖子。随书源代码包含 solrexchange.xml 文件，

无须下载整个 Stack Exchange 数据文件就可以测试这个配置。如果你决定下载完整的 Stack Exchange 数据文件，请使用 Post.xml 替换 solrexchange.xml，可以从社区的 7z 文件中解压提取 Post.xml 文件。

在 Solr 运行过程中，打开 Solr 管理界面的数据导入页面，如图 C.1 所示，开始导入 Stack Exchange 文件，与之前导入维基百科文档过程一样。

DIH 不仅能导入 XML 文档，还能处理网站、数据库及其他数据来源。要了解关于 DIH 功能的更多信息，参见第 12 章。

# Solr 实战

无论是处理大（小）数据、管理文档还是构建网站，能够快速搜索内容并发现其中的含义是非常重要的。Apache的Solr就是这样一把利器：部署待命、基于Lucene、开源的全文搜索引擎。Solr可以扩展到多个服务器，支持实时查询和数十亿文档的数据分析。

《Solr实战》教你使用Apache的Solr实现可扩展的搜索。这是一本易于阅读的指南，很好地平衡了概念讨论与实践操作，展示了如何实现Solr的全部核心功能。通过阅读本书，读者可以掌握文本分析、分面搜索、搜索结果高亮、搜索结果分组、查询建议、多语种搜索、高级地理空间与数据操作，以及相关度调整等主题。

## 本书内容包括

- 如何用Solr来应对大数据
- 丰富的现实世界的例子
- Solr作为一种NoSQL数据存储
- 多语种、数据与相关度的高级技巧

阅读本书需要具备Java与数据库的基础知识，但不需要具备Solr或Lucene的先修知识。

## 作者简介

Trey Grainger是CareerBuilder公司的工程总监。Timothy Potter是LucidWorks公司工程组的资深成员。两位作者都在从事Solr的可扩展性和可靠性、推荐引擎及大数据分析技术等方面的工作。

## 译者简介

范炜，四川大学信息管理技术系副教授，情报学硕士生导师。主要从事信息组织与检索方面的教学科研工作。参编的书籍有《信息管理导论（第3版）》和《信息组织（第3版）》，进行技术审校的书籍有《Web信息架构（第2版）》和《搜索模式》。中国图书馆学会信息组织专业委员会委员，国际十进制分类法UDC咨询委员会委员，ASIS&T会员，国际知识组织学会ISKO会员。



MANNING



策划编辑：符隆美  
责任编辑：徐津平  
封面设计：王乐

你需要的知识和技术都在这里。

——引自Solr的创造者Yonik Seeley  
为本书所作的推荐序

一本可读性好的、马上可以用起来的好书。

——John Viviano, InterCorp, Inc.

Solr指南书……初学者和专家必备的权威资源。

——Scott Anthony, Business Instruments

深厚的技术知识与现实世界经验恰到好处地组合在了一起。

——Alexandre Madurell, Pikel, Inc.

上架建议：搜索引擎/程序开发

ISBN 978-7-121-31165-9



9 787121 311659 >

定价：129.00元